MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY
«KHARKIV POLYTECHNIC INSTITUTE»

S. S. Bulba, V. O. Brechko, N. G. Kuchuk

ALGORITHMS AND DATA STRUCTURES

**Educational-methodical manual
for students of specialty 123
«Computer Engineering».**

Recommended by the editorial and publishing board of the university, protocol No. 1 dated February 15, 2024.

Kharkiv
NTU «KhPI»
2024

UDC 004
B 90

Reviewers:
O. HRYB, Doctor of Technical Sciences, Professor of NTU "KhPI"
M. SKULYSH, Doctor of Technical Sciences, Professor of National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute"

Bulba S. S.
B 90 Algorithms and Data Structures : teaching methodological manual / S. S. Bulba, V. O. Brechko, N. G. Kuchuk. – Kharkiv : NTU «KhPI», 2024. – 140 p.

The main data structures are presented, issues of algorithm analysis are discussed, and methods of achieving maximum asymptotic algorithm performance are covered. The material is illustrated with practical examples, and each section includes necessary problems and exercises.
For students majoring in 123 "Computer Engineering".

Table 19. Figures 14. Bibliography 8.

UDC 004

INTRODUCTION

> “A good programmer should know everything that has been written before them, only then will they be able to write good programs.”
>
> Bentley
>
> “Bad programmers worry about the code. Good programmers worry about data structures and their relationships”
>
> Linus Torvalds, creator of Linux

The words of J. Bentley and L. Torvalds can be an epigraph to the discipline "algorithms and data structures". They are about the need to study a large number of already known algorithms and data structures. Nowadays, it is not necessary to write a program for all tasks that arise during work. There are cases when it is enough to use an already existing program or combine several well-known programs. But this is rather an exception to the rule.

In general, the development of a software product is a complex process, in which the actual programming is only one of its stages. Therefore, it is not enough to know any programming language. It is necessary to have experience in algorithmic thinking and optimal selection of data structures. Ultimately, the complexity and quality of the developed software product depends on this. Not every programmer can easily develop high-quality software products that would meet the requirements for speed, memory resources, reliability, data confidentiality, testing, extensibility, interaction with other programs, etc.

As part of the discipline "algorithms and data structures", it is planned to pay attention to the main data structures, consider the issues of algorithm analysis, methods of reducing memory requirements, and achieving good asymptotic performance of algorithms.

The goal of the discipline is to study a set of important and useful algorithms; special attention will be paid to fundamental algorithms, which must be known and be able to apply them in practice. To achieve the goal, the main attention is paid to the implementation of algorithms and data structures in the practical classes and to the determination of their comparative characteristics by conducting a large number of experiments.

CONTENT

PRACTICAL ASSIGNMENT 1. ESTIMATION OF THE LABOR INTENTION OF THE ALGORITHM

GOAL: MASTERING ANALYTICAL METHODS OF ANALYZING THE COMPLEXITY OF COMPUTING ALGORITHMS.

General Information. An algorithm is a precise sequence of actions, which in a finite number of steps leads from an arbitrary input data (or from some set of input data) to the achievement of a result that is completely determined by these input data.

It is accepted to analyze algorithms according to the following parameters:

- labor intensive;

- complexity;

- productivity.

Computational complexity is a function of the dependence of the amount of work performed by some algorithm on the size of the input data. Algorithm complexity can be estimated in bits, bytes, or the number of symbols of a certain language. A preliminary assessment of the complexity of the algorithm and the amount of data is performed by an expert, and the exact values can be known only after the programming is completed. The complexity of the algorithm and the amount of data characterize the task's need for RAM and external memory.

The labor intensity of the algorithm is understood as the amount of computational work required for its implementation. Labor intensity characterizes the time required to implement the algorithm on some set of technical means. Usually, labor intensity is estimated by the number of processor operations and input-output operations. It should be noted right away that the labor intensity of the algorithm is generally a random value and depends on the input data. Therefore, the labor intensity of the algorithm can be determined only approximately in terms of probability theory: mathematical expectations, variance, etc.

The labor intensity of the algorithm can be roughly defined as the average number of processor operations (Θ) required to execute the algorithm one time. To simplify calculations, input-output operations will not be taken into account. But if necessary, the set of parameters characterizing the labor intensity of the

algorithm can be expanded.

The initial information for calculating the complexity of the algorithm will be taken from the scheme of the algorithm, which consists of operators V_1, V_2,..., V_{k-1} and according to which the program is developed.

The average number of calls to operators V_1, V_2, . . ., V_{k-1} is denoted as n_1, n_2,..., n_{k-1}, відповідно. Each of the V_i operators *(i=1, 2, ..., k-1)* is characterized by the average number of processor operations k_i , performed in the V_i operator. Then the labor intensity of the algorithm can be estimated by the following formula:

$$\Theta = \sum_{i=1}^{k-1} n_i q_i \qquad (1.1)$$

where Θ is the average number of processor operations performed during one run of the algorithm; q_i is the average number of processing operations of operator V_i ; n_i is the average number of calls to operator V_i ; k is the number of operators in the algorithm.

To determine the average number of calls n_i to the operator V_i *(i=1,2,...,k-1)* generally, the following assumptions are made:

– the probability of execution of operator V_j after operator V_i is equal to P_{ij} and is a constant value;

– the probability P_{ij} depends only on the operator V_i, but is in no way related to the way of getting into the operator V_j, that is, it does not depend on the history of the computing process. Therefore, the condition is fulfilled for all operators:

$$\sum_{i=1}^{k-1} P_{ij} = 1$$

If the above assumptions are met, the computational process is markov with states *S1, S2,..., Sk.* At the same time, the operators *V1, V2,..., Vk-1* correspond to the states *S1, S2,..., Sk-1*. The *Sk* state corresponds to the finite vertex of the algorithm graph and is absorbing, that is, when the *Sk* state is reached, the process stops. States *S1, S2,..., Sk-1* are called irreversible, because the process necessarily leaves them.

There are several ways of estimating the complexity of the algorithm; it would be considered two of them:

- the method of the of Markov chains theory;
- the network method.

1. Estimation of the complexity of algorithms using the methods of Markov chain theory

To determine the average number of processor operations performed in one run of the program, it is necessary to write the graph of the algorithm in the form of a stochastic matrix P.

The elements of matrix P are the transition probabilities from state i to state j, which are shown on the graph of the algorithm. In the general case, it is possible to write (summation by column of the stochastic matrix):

$$n_i = \sum_{j=0}^{k-1} p_{ji} n_{j}, \quad (n_0 = 1, i = 1, 2, ..., k\text{-}1) \tag{1.2}$$

The last entry is a system of linear algebraic equations, the solution of which will give the average number of calls to the operators under investigation. To verify the correctness of solving the system of linear equations, one can use the obvious identity:

$$n_k = \sum_{j=0}^{k-1} p_{jk} n_j = 1, \quad \text{if} \quad n_0 = 1 \tag{1.3}$$

2). Network approach to algorithm complexity estimation.

The network approach is convenient for manual analysis and allows to calculate the average, minimum and maximum labor intensity of algorithms on graphs that do not contain loops. If there are loops in the graph of the algorithm, the latter should be replaced by operators with equivalent labor intensity. If there are nested loops in the graph of the algorithm, then first you should replace the inner loops with equivalent operators, and then move on to replacing the outer loops. Labor intensity calculations of the algorithm are performed according to the above method.

Let us denote by $\boldsymbol{p_c}$ the probability of closing the cycle, i.e. the probability of transition along the arc from the end of the cycle to its beginning. Then,

according to expression (1.3), we can write:

$$n_c = 1 + p_c n_c, \quad (1.4)$$

where n_c is the average number of cycle repetitions.

From expression (1.4) there is:

$$n_c = \frac{1}{1-p_c} \quad (1.5)$$

Then the average number of processor operations of the cycle is equal to:

$$q_c = n_c q_{mc},$$

where q_{mc} is the average labor intensity of the loop body, determined by the formula (1.1), but only the operators constituting the loop body are taken into account; q_c is the average labor intensity of cycle.

Typical task

Calculate the labor intensity of the algorithm, the scheme of which is shown in Fig. 1.1. Use the network method and the Markov chain theory method.

To calculate the labor intensity of the algorithm, it is necessary to know the probabilities of transitions from logical vertices with a single (*true*) value of the logical condition. If the corresponding probability is denoted by p, then the probability of exiting the logical vertex with the zero logical value of the condition (*false*) being checked will be equal to *1 – p.*

For further calculations, it is advisable to depict the algorithm scheme in the form of an algorithm graph. To do this, you need to renumber all the operators of the algorithm scheme. In logical operators, instead of logical conditions "1" and "0", it is necessary to write the probability corresponding to this output. The probability of exit from the operator's room in fig. 1.2.

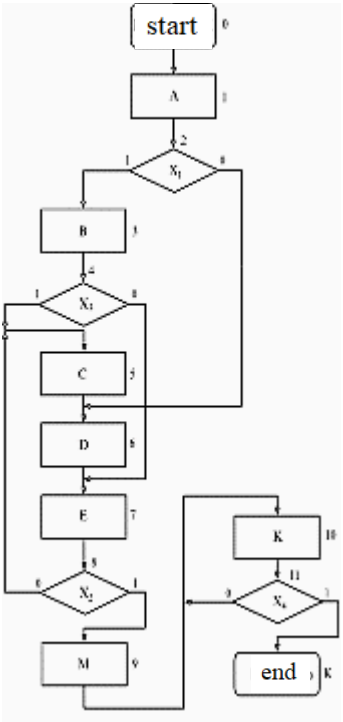


Figure 1.1 – Scheme of the algorithm

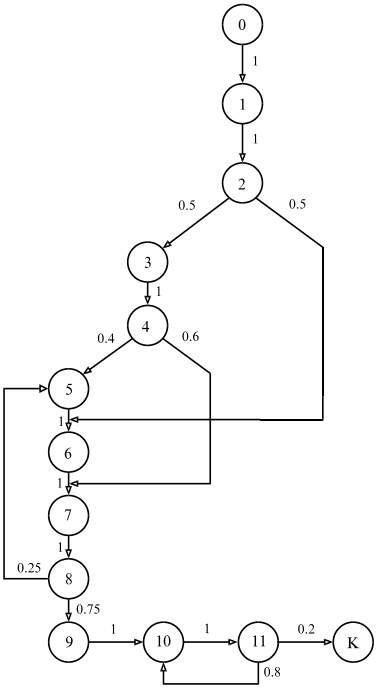


Figure 1.2 – Graph of the algorithm

The graph of the algorithm can be significantly simplified if the complexity of executing logical vertices is not significant compared to the complexity of executing operator vertices. Then the states corresponding to the logical vertices can be merged with the previous states corresponding to the operator vertices. In the given example, states (1, 2), (3, 4), (7, 8), (10, 11) can be merged. After renumbering the vertices, a minimized graph of the algorithm is obtained, shown in Fig. 1.3. It is clear that with sufficient experience it would be possible to reach it immediately from the algorithm diagram shown in fig. 1.1.

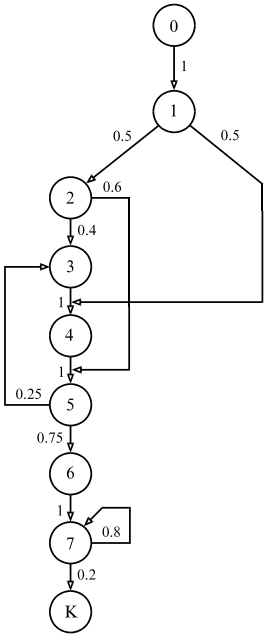


Figure 1.3 – The minimized graph of the algorithm

1. Evaluation of the labor intensity of algorithms by the methods of the theory of Markov chains

To determine the average number of processor operations performed in one run of the program, the graph of the algorithm should be written in the form of a stochastic matrix P. For the graph (Fig. 1.3), the stochastic matrix is given in the table. 1.1. In the table, the names of the initial states (from which the transition is performed) are indicated in the first column, and the names of the final states (for which the transition is performed) are indicated in the first row.

Table 1.1 – Stochastic matrix

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_k
S_0	1	0	0	0	0	0	0	0
S_1	0	0.5	0	0.5	0	0	0	0
S_2	0	0	0.4	0	0.6	0	0	0

S_3	0	0	0	1	0	0	0	0
S_4	0	0	0	0	1	0	0	0
S_5	0	0	0.25	0	0	0.75	0	0
S_6	0	0	0	0	0	0	1	0
S_7	0	0	0	0	0	0	0.8	0.2

It follows from the stochastic matrix, for example, that the process can appear in state S_4 when transitioning from S_1 with a probability of 0.5 or when transitioning from state S_3 with a probability of 1. Provided that the average numbers of visits to vertices V_1 and V_3 are known, then the number of visits to the vertex V_4 will accordingly be:

$$n_4 = p_{14}\, n_1 + p_{34}\, n_3 = 0.5n_1 + n_3 .$$

From the stochastic matrix, we similarly determine the number of calls to all vertices of the graph:

$n_1 = 1 \cdot n_0 = 1 \cdot 1 = 1$; $n_2 = 0.5 \cdot n_1 = 0.5 \cdot 1 = 0.5$;

$n_3 = 0.4 \cdot n_2 + 0.25 \cdot n_5 = 0.2 + 0.25 \cdot n_5$;

$n_4 = 0.5 \cdot n_1 + n_3 = 0.5 + n_3$; $n_5 = 0.6 \cdot n_2 + n_4 = 0.3 + n_4$;

$n_6 = 0.75 \cdot n_5$; $n_7 = n_6 + 0.8 \cdot n_7$.

As a result of solving the system of equations, the following is obtained:

$n_1 = 1$; $n_2 = 0.5$; $n_3 = 0.533$; $n_4 = 1.033$; $n_5 = 1.333$; $n_6 = 1$; $n_7 = 5$.

Checking the correctness of the solution of the system of linear equations

For the specified algorithm, the check is performed as follows:

$$n_k = 0.2 \cdot n_7 = 0.2 \cdot 5 = 1.$$

Since entry into the final state n_k is equal to 1, the calculation is correct. According to formula (1.1), assuming that qi for all V_i operators is 1000 operations, we calculate Θ:

$$\Theta = (1 + 0{,}5 + 0.533 + 1.033 + 1.333 + 1 + 5) \cdot 1000 = \mathbf{10399}$$ operations.

This method is universal (in the case of a Markov process), but requires more time when the stochastic matrix is large.

2. A network approach to the assessment of the labor intensity of algorithms.

The network approach is convenient for manual analysis and allows to calculate the average, minimum and maximum labot intensity of the algorithm on graphs that do not contain loops. If the graph contains loops, first you need to determine the number of calls to each of the loops and the corresponding number of processor operations. Then rebuild the graph in which each of the loops is replaced by the corresponding state with the calculated number of processor operations.

In the algorithm in fig. 1.3, which is considered, there are two cycles. The first contains the operators V_3, V_4, V_5, and the second consists only of the operator V_7. The first loop is complicated by inputs inside the loop from operators V_1 and V_2. In order to get rid of inputs to the cycle not due to the start of the V_3 cycle, let's make elementary transformations of the algorithm graph. The equivalent graph after obvious transformations is shown in fig. 1.4. From fig. 1.4 it can be seen that operators V_4 and V_5, which were used to enter the body of the loop not through its beginning, are placed separately under the numbers 4' and 5'.

Let's determine the number of cycle repetitions from expression (1.5):

$$n_{C1} = \frac{1}{1-0.25} = 1.33; n_{C2} = \frac{1}{1-0.8} = 5;$$

where n_{c1}, n_{c2} is the number of repetitions of the first and second loops.

The graph of the algorithm without loops is shown in Fig. 1.5.

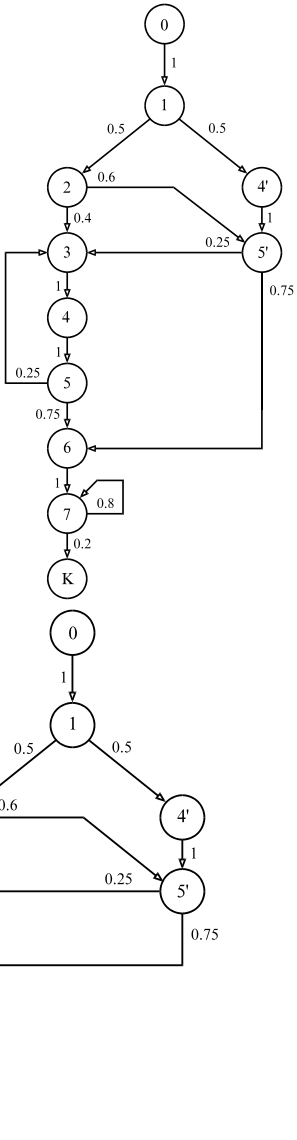


Figure 1.4 – Equivalent algorithm graph

Figure 1.5 – Algorithm graph without loops

Let's calculate the average number of processor operations for loops C1 and C2:

$q_{C1} = 1000 \cdot 3 \cdot 1.333 = 3999$ op.;

$q_{C2} = 1000 \cdot 5 = 5000$ op.

Using formula (1.2) for fig. 1.5, it is possible to write the following:

$n_0 = 1;$ $n_1 = 1 \cdot n_0 = 1;$

$n_2 = 0.5 \cdot n_1 = 0.5;$ $n_{4'} = 0.5 \cdot n_1 = 0.5;$

$n_{5'} = n_{4'} + 0.6n_2 = 0.5 + 0.6 \cdot 0.5 = 0.8;$

$n_{3'} = 0.4n_2 + 0.25n_{5'} = 0.4 \cdot 0.5 + 0.25 \cdot 0.8 = 0.4;$

$n_6 = n_{3'} + 0.75n_{5'} = 0.4 + 0.75 \cdot 0.8 = 1;$ $n_{7'} = 1 \cdot n_6 = 1.$

Due to the absence of loops, the system is easily solvable. Time spent on analysis can be further reduced if effective numbering of algorithm graph states is introduced. The last thing is that the number of the vertex should be such that the arcs entering this vertex start at vertices with smaller numbers. If, after such numbering, the equations are written in ascending order of their numbers, then they will be solved immediately, because the unknowns with smaller numbers will have already been calculated. In the given example, the numbering of the vertices does not meet the specified requirement, but the equations were written in the order of their solution.

The above approach allows you to calculate the average labor intensity of the algorithm.

According to the formula (1.1), under the condition that q_i for all V_i operators is 1000 operations, we calculate Θ.

$\Theta = 1000 + 500 + 0.4 \cdot 3999 + 500 + 800 + 1000 + 5000 =$ **10399** op.

Conclusions: the results of calculating the average number of processor operations determined using the stochastic matrix and the network approach algorithm coincided.

To estimate the maximum and minimum complexity of the algorithm, it is necessary to go through all the possible paths leading from the initial vertex of the graph of the algorithm to the final one, and to select from them such paths that give the maximum and minimum complexity.

If there are loops in the graph of the algorithm, the minimum and maximum complexity is determined for the body of the cycle. The minimum and maximum number of repetitions of the cycle are determined and the corresponding operators equivalent in terms of labor intensity are formed.

Let's estimate the minimum and maximum number of processor operations for the graph of the algorithm shown in Fig. 1.5. For simplicity, let's assume that all operators have a labor intensity equal to 1000 processor operations. By A_i and B_i, we denote, accordingly, the minimum and maximum number of processor operations that will occur at the time the process exits from the ith vertex of the graph.

Calculation results.

1. Minimum number of processor operations

$A_0 = 0$;

$A_1 = \min(A_0) + 1000 = 1000$;

$A_2 = \min(A_1 \cdot 0.5) = 500 + 1000 = 1500$;

$A_{4'} = \min(A_1 \cdot 0.5) = 500 + 1000 = 1500$;

$A_{5'} = \min(A_2 \cdot 0.6, A_{4'}) = \min(1500 \cdot 0.6 = 900, 1500) = 900 + 1000 = 1900$;

$A_{3'} = \min(A_2 \cdot 0.4, A_5 * 0.25_{'}) = \min(1500 \cdot 0.4 = 600, 1900 \cdot 0.25 = 475) = 475 + + 3999 = 4474$;

$A_6 = \min(A_{3'}, A_{5'} \cdot 0.75_{'}) = \min(4474, 1900 \cdot 0.75 = 1425) = 1425 + 1000 = 2425$;

$A_7 = \min(A_6) + 5000 = 2425 + 5000 =$ **7425**;

2. The maximum number of processor operations

$B_0 = 0$;

$B_1 = \max(B_0) + 1000 = 1000;$

$B_2 = \max(B_1 \cdot 0.5) + 1000 = 500+1000 = 1500;$

$B_{4'} = \max(B_1 \cdot 0.5) + 1000 = 500+1000 = 1500;$

$B_{5'} = \max(B_2 \cdot 0.6, B_{4'}) = \max(1500 \cdot 0.6=900, 1500) = 1500 + 1000 = 2500;$

$B_{3'} = \max(B_2 \cdot 0.4, B_{5'} \cdot 0.25) = \max(1500 \cdot 0.4=600, 2500 \cdot 0.25= 625) = 625 + + 3999 = 4624;$

$B_6 = \max(B_{3'}, B_5 \cdot 0.75') = \max(4624, 2500 \cdot 0.75=1875) = 4624 + 1000 = 5624;$

$B_7 = \max(B_6) + 5000 = 5624 + 5000 =$ **10624**.

Conclusions: the minimum number of processor operations is less than the average number, and the maximum number is more than the average number of processor operations. So, the calculation is correct.

Individual tasks

1. From the table 1.2 choose the logical scheme of the algorithm (LSA) according to the variant.

In LSA, the symbols "Begin" and «End» correspond to the initial and final operators of the algorithm. The symbols A, B, C, D, E, K, M denote the functional operators of the algorithm. Symbols x1, x2, x3, x4 denote logical conditions. If the logical condition is equal to one, then the following one operator in LCA is executed. If the logical condition is equal to zero, then the transition is carried out by the arrow with the corresponding index, in this case, the arrow is pointing up at the logical condition (for example, $\uparrow^i$). At the place of transition from the logical condition, the arrow points down – $\downarrow^i$.

For example, for the scheme of the algorithm shown in fig. 1.1, LSA has the form:

Begin. $Ax_1\uparrow^1 Bx_3\uparrow^3\downarrow^2 C\downarrow^1 D\downarrow^3 Ex2\uparrow^2 M\downarrow^4 Kx4\uparrow^4$ End.

From the table 1.3 choose the transition probabilities under single logical conditions.

2. According to LSA, build a graphical scheme of the algorithm, a graph of the algorithm and a minimal graph of the algorithm.

3. Determine the labor intensity of the algorithm using the methods of the theory of Markov chains.

4. Determine the average labor intensity of the algorithm using a network approach. First, if there are loops in the algorithm, determine the average complexity of the loops.

5. Calculate the minimum and maximum labor intensity of the algorithm.

6. Analyze the obtained results.

Table 1.2 – Descriptions of algorithm schemes

Variant	the logical scheme of the algorithm
1	Begin. $Ax_1\uparrow^1 Bx_2\uparrow^2\downarrow^3 Cx_3\uparrow^3 D\downarrow^2 E\downarrow^4 Mx_4\uparrow^4 K\downarrow^1$ End.

2	Begin. $Ax_2\uparrow^2Bx_3\uparrow^3Cx_1\uparrow^1DE\downarrow^1\downarrow^3M\downarrow^2x_4\uparrow^4K\downarrow^4$ End.
3	Begin. $Ax_2\uparrow^2B\downarrow^3\downarrow^2Cx_3\uparrow^3\downarrow^1Dx_1\uparrow^1Ex_4\uparrow^4MK\downarrow^4$ End.
4	Begin. $A\downarrow^2\downarrow^1Bx_1\uparrow^1Cx_2\uparrow^2Dx_4\uparrow^4\downarrow^3Ex_3\uparrow^3M\downarrow^4K$ End.
5	Begin. $\downarrow^1Ax_1\uparrow^1Bx_2\uparrow^2Ex_3\uparrow^3C\downarrow^2\downarrow^3Mx_4\uparrow^4D\downarrow^4K$ End.
6	Begin. $A\downarrow^4Bx_2\uparrow^2Cx_3\uparrow^3D\downarrow^2\downarrow^3Ex_4\uparrow^4Mx_1\uparrow^1K\downarrow^1$ End.
7	Begin. $Ax_2\uparrow^2Cx_1\uparrow^1Dx_3\uparrow^3\downarrow^1E\downarrow^2\downarrow^3\downarrow^4Kx_4\uparrow^4MB$ End.
8	Begin. $Dx_1\uparrow^1Ex_2\uparrow^2B\downarrow^2\downarrow^3Ax_4\uparrow^4Mx_3\uparrow^3\downarrow^4CK\downarrow^1$ End.
9	Begin. $x_1\uparrow^1A\downarrow^1E\downarrow^2Bx_2\uparrow^2C\downarrow^4Dx_3\uparrow^3K\downarrow^3Mx_4\uparrow^4$ End.
10	Begin. $x_1\uparrow^1A\downarrow^4\downarrow^2Bx_2\uparrow^2Cx_3\uparrow^3E\downarrow^3Dx_4\uparrow^4K\downarrow^1M$ End.
11	Begin. $\downarrow^3Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2D\downarrow^1Ex_3\uparrow^3Kx_4\uparrow^4M\downarrow^4$ End.
12	Begin. $x_4\uparrow^4Ax_1\uparrow^1B\downarrow^2Cx_2\uparrow^2D\downarrow^1Ex_3\uparrow^3K\downarrow^4M\downarrow^3$ End.
13	Begin. $Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2\downarrow^1D\downarrow^3\downarrow^4Ex_4\uparrow^4Mx_3\uparrow^3$ K End.
14	Begin. $Ax_1\uparrow^1Bx_2\uparrow^2C\downarrow^2D\downarrow^3Ex_3\uparrow^3\downarrow^1Kx_4\uparrow^4M\downarrow^4$ End.
15	Begin. $\downarrow^4Ax_1\uparrow^1B\downarrow^1Cx_2\uparrow^2Dx_3\uparrow^3\downarrow^2K\downarrow^3Mx_4\uparrow^4$ End.
16	Begin. $\downarrow^1Ax_1\uparrow^1B\downarrow^4\downarrow^2Cx_2\uparrow^2Dx_4\uparrow^4EKx_3\uparrow^3M\downarrow^3$ End.
17	Begin. $\downarrow^1Ax_1\uparrow^1Bx_3\uparrow^3C\downarrow^3Dx_4\uparrow^4E\downarrow^2Kx_2\uparrow^2M\downarrow^4$ End.
18	Begin. $\downarrow^4Ax_4\uparrow^4Bx_1\uparrow^1C\downarrow^1Dx_3\uparrow^3Ex_2\uparrow^2K\downarrow^2M\downarrow^3$ End.
19	Begin. $x_1\uparrow^1A\downarrow^3Bx_2\uparrow^2Cx_4\uparrow^4D\downarrow^2Ex_3\uparrow^3\downarrow^1KM\downarrow^4$ End.
20	Begin. $\downarrow^4Ax_1\uparrow^1B\downarrow^1C\downarrow^2\downarrow^3DEx_2\uparrow^2Kx_3\uparrow^3Mx_4\uparrow^4$ End.
21	Begin. $x_1\uparrow^1A\downarrow^3Bx_2\uparrow^2Cx_4\uparrow^4D\downarrow^4E\downarrow^2K\downarrow^1Mx_3\uparrow^3$ End.
22	Begin. $x_2\uparrow^2A\downarrow^3Bx_1\uparrow^1C\downarrow^2Dx_4\uparrow^4E\downarrow^1\downarrow^4KMx_3\uparrow^3$ End.
23	Begin. $x_1\uparrow^1Ax_2\uparrow^2B\downarrow^2\downarrow^1Cx_4\uparrow^4D\downarrow^3Ex_3\uparrow^3KM\downarrow^4$ End.
24	Begin. $x_1\uparrow^1A\downarrow^2B\downarrow^1Cx_2\uparrow^2D\downarrow^3Ex_3\uparrow^3\downarrow^4Mx_4\uparrow^4K$ End.
25	Begin. $x_3\uparrow^3Ax_2\uparrow^2\downarrow^1Bx_1\uparrow^1C\downarrow^2\downarrow^3D\downarrow^4Ex_4\uparrow^4MK$ End.

Table 1.3 – Probabilities of transition (by X=1)

№	P1	P2	P3	P4
1	0.1	0.3	0.6	0.9
2	0.2	0.2	0.7	0.8
3	0.3	0.1	0.8	0.7
4	0.4	0.2	0.9	0.6
5	0.5	0.3	0.8	0.5
6	0.6	0.4	0.7	0.4
7	0.7	0.5	0.6	0.3
8	0.8	0.6	0.5	0.2
9	0.9	0.7	0.4	0.1
10	0.8	0.8	0.3	0.2
11	0.7	0.9	0.2	0.3
12	0.6	0.8	0.1	0.4
13	0.5	0.7	0.2	0.5
14	0.4	0.6	0.3	0.4
15	0.3	0.5	0.4	0.3
16	0.2	0.4	0.5	0.2
17	0.1	0.3	0.6	0.1
18	0.2	0.2	0.7	0.2
19	0.3	0.1	0.8	0.3
20	0.4	0.2	0.9	0.4

Table 1.4 – Number of processor operations in operators (in thousands)

№	A	B	C	D	E	M	K
1	1	2	3	4	5	6	7
2	8	9	8	7	6	5	4
3	3	2	1	2	3	4	5
4	6	7	8	9	8	7	6
5	5	4	3	2	1	2	3
6	4	5	6	7	8	9	8
7	7	6	5	4	3	2	1
8	2	3	4	5	6	7	8
9	9	8	7	6	5	4	3
10	2	1	2	3	4	5	6
11	7	8	9	8	7	6	5
12	4	3	2	1	2	3	4
13	5	6	7	8	9	8	7
14	6	5	4	3	2	1	2
15	5	3	4	2	1	5	4
16	3	4	5	6	7	8	9
17	1	3	5	7	9	8	6
18	4	2	2	4	6	8	9
19	7	5	3	1	1	3	5
20	7	9	8	6	4	2	2

21	0.5	0.3	0.8	0.5
22	0.6	0.4	0.7	0.6
23	0.7	0.5	0.6	0.7
24	0.8	0.6	0.5	0.8
25	0.9	0.7	0.4	0.9

21	4	6	8	9	7	5	3
22	1	4	7	9	8	7	6
23	5	4	3	9	7	6	8
24	9	4	3	7	8	8	6
25	2	4	9	7	8	6	2

Checkpoint questions

1. What is an "algorithm"? Give a definition.
2. What are the mandatory properties of the algorithm?
3. What is the labor intensity of the algorithm?
4. Give the classification of algorithms according to the function of labor intensity.
5. What is the complexity of the algorithm?
6. How is the complexity of the algorithm determined, and in what cases is this assessment required?
7. How is the labor intensity of the algorithm determined, and for what purpose is this value calculated?
8. Why is the labor intensity of the algorithm usually a random value?
9. What parameters can be used to characterize the labor intensity of the algorithm?
10. How to perform effective numbering of states in the algorithm graph for network analysis?
11. What is a stochastic matrix?
12. What is a Markov process?
13. How to determine the probability of exit from the logical vertice? Give

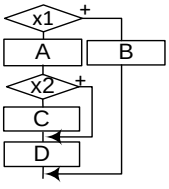
examples.

14. Why is the assumption made about the computational process as Markov when estimating the labor intensity of the algorithm?

15. Is there a difference in the estimation of the average labor intensity when using the methods of the theory of Markov chains and the network approach? Explain it.

16. Determine the labor intensity of the given fragment of the algorithm.

The table shows the number of operations in the calculation blocks and the probability of positive results in the selection blocks.



A	B	C	D	x1	x2
40	20	60	50	0,1	0,3

PRACTICAL ASSIGNMENT 2. RECURSIVE AND ITERATIVE ALGORITHMS

***GOAL**:* TO ACQUIRE SKILLS AND PRACTICAL EXPERIENCE IN DEVELOPING RECURSIVE PROGRAMS.

Preparatory work:

- iterative algorithms;

- recursive algorithms.

General Information

An object that is partially formed or defined by itself is called recursive.

A recursive function can describe an infinite calculation, and such a function will not contain explicit repetitions. Therefore, it is necessary to ensure the end of its work. The most reliable way is to enter any parameter (value) into it, let's call it n, and when recursively referring to P, set n – 1 as a parameter. If in this case n > 0 is used as a condition, then the termination is guaranteed. This provision can be expressed in two schemes:

$$P(n) \equiv if\ n>0,\ then\ P[S,P(n-1)]$$

or

$$P(n) \equiv P[S,\ if\ n>0,\ then\ P(n-1)].$$

Recursion is an effective program control mechanism. When a method calls itself, the following is written to the system stack: return address, register values, parameters, and new local variables. As a result, the method code is executed with these new parameters and variables from the very beginning. When a method is called recursively, a new copy of the method is not created, only its parameters are used. And when returning from each recursive call, old local variables and parameters are pushed from the stack, register values and addresses are read. As a result, program execution resumes from the point of the method call.

Recursive versions of many procedures can run much more slowly than their iterative equivalents due to the additional system resource costs of multiple

method calls. If there are a lot of such calls, the system stack may be full. And since the parameters and local variables of the recursive method are stored in the system stack and a new copy of them is created with each new call to this method, at some point the stack may be depleted, leading to an exceptional situation.

The main advantage of recursion is that it allows you to implement some algorithms more clearly and simply than the iterative method. An example can be a quick sort algorithm, which is more difficult to implement by an iterative method than by a recursive one.

Typical tasks

Task 1. Calculate the value of the function $f = x^n$.

Program code

```
#include <iostream>
#include <conio.h>
using namespace std;

int power( int x, int n); //  Returns x to the power of n (n>=0)

int main(void)
{
    for ( int n = 0; n< 4; n++)
       cout << " 3 to the power of "<< n << " is equal to "<<power(3, n)<< endl;
    return 0;
}
int power( int x, int n)
{   if ( n <0)

    {    cout <<" Invalid argument to function power.\n";

         exit(1);

    }

    if (n >0 )  return power(x, n-1)*x ;

    else return 1 ;// n == 0

}
```

The result of the program:

3 to the power of 0 is equal to 1

3 to the power of 1 is equal to 3

3 to the power of 2 is equal to 9

3 to the power of 3 is equal to 27

A classic example of recursion is calculating the factorial of a number.

Task 2. Calculate the factorial ***F = n!.***

Program code

```
#include<iostream>
using namespace std;

 int FactR(int n) // recursive method
 {    int result;
      if (n==1)  return 1;
      result =  FactR(n-1) * n;
      return result;
 }
  int FactI(int n)  // iterative method
 {   int result = 1;
     for (int t=1; t<=n; t++)  result *= t;
     return result;
 }

  void main()
 {
        cout << "Recursive method" << "5!="<< FactR(5) << endl;

              cout << "Iterative method" << "5!="<< FactR(5) << endl;

 }
```

Regarding the ***FactR*** recursive method, it can be noted that its action is organized according to the following principle: if the method is called with an argument of 1, it returns the value 1. Otherwise, it returns the product ***FactR(n-1) * n.*** To calculate this product, you need to call ***FactR*** again, but with an argument of ***n-1***. This process is repeated until ***n*** is equal to ***1***, after which the values obtained from previous method calls will begin to return. For example, When the factorial of the number ***2*** is calculated, the first call to the ***FactR*** method is called again with the argument ***1***. The value ***1*** is returned from this call, which is then multiplied by ***2***. The result will be ***2***.

The result of the program:

Recursive method 5! = 120

Iterative method 5! = 120

Task 3. Display the string in reverse order.

Program code

```
#include<iostream>
#include<string>
using namespace std;

 void DisplayRev(string str)
 {   if (str.length() > 0)
           DisplayRev(str.substr(1, str.length()-1));
     else  return;
     cout << str[0];
 }

  void main()
 {
     string s = "This is test";
     cout << " Initial line: "<<  s << endl;
     cout << " Inverted line: ";
          DisplayRev(s);
    cout << endl;
 }
```

The result of the program:

```
Initial line: This is test

Inverted line: tset si sihT
```

Whenever the ***DisplayRev*** method is called, it checks the character string given by the ***str*** argument. If the string length is not zero, the ***DisplayRev*** method is called recursively with a new string that is one character shorter than the input string. This process is repeated until a zero-length string is passed to the method. After that, the mechanism of all recursive calls to the ***DisplayRev*** method will begin to unwind in reverse order. When returning from each such call, the first character of the line given by the argument ***str*** is displayed on the screen. Thus, the entire line is displayed in reverse order.

Individual tasks

Develop recursive and iterative algorithms for solving individual tasks.

Determine and compare the time of execution of the corresponding

functions, draw conclusions.

1. The Ackerman function A is defined for all positive (>0) integer arguments m and n as follows:

$$A(0,n) = n + 1;$$

$$A(m,0) = A(m - 1,1); \qquad (m > 0)$$

$$A(m,n) = A(m - 1, A(m, n - 1)); \qquad (m > 0, n > 0)$$

Calculate the value of the function A(m,n) for m and n entered from the keyboard.

2. Find all *n!* rearrangements for *n* elements $a_1 \ldots a_n$ and output them to the screen. Enter the number *n* and the numbers that are rearranged from the keyboard.

4. A text (character string) is given. Check whether a string of characters starting with index *start* and ending with index *end* is a palindrome.

5. The coefficients forming Pascal's triangle are defined as follows:

$$C(n,0) = 1;$$

$$C(n,n) = 1; \qquad (n > 0)$$

$$C(n,k) = C(n - 1,k - 1) + C(n - 1,k); \qquad (n > 0, m > 0)$$

Develop a program that builds Pascal's triangle for a given *n*.

6. The algorithm for converting the number *N* from one number system to another (B) consists in multiple division by B. If $N = d_{n-1}\, d_{n-2}\, d_{n-3} \ldots d_1\, d_0$, then the sequence of remainders from the division is in the order d0 … dn-1 gives the digits of the conversion result. Develop a function that converts the number *N* into the number system B, assume the condition *B<=10*.

7. For a given n, determine all Fibonacci numbers from 0 to n. Fibonacci numbers are defined as follows:

$$fib_{n+1} = fib_n + fib_{n-1}, \text{ для } n > 0; \quad fib_0 = 0; fib_1 = 1;$$

8. Develop a program to determine the payment amounts for a loan of $100,000 at an annual interest rate of 10% for 20 years.

9. Compute the root of the equation $f(x) = x^3 - 2x^2 - 3x + 10$ with a given precision in the interval $a <= x <= b$ using the bisection method.

10. The bisection method is defined as follows: if *f(a)* and *f(b)* have different signs, then there is a root between *a* and *b*. The midpoint $m = (a+b) / 2$ is computed. If $f(m) = 0$, then the root is found. Otherwise, the intervals $a <= x <= m$ and $m <= x <= b$ are checked, and further actions are performed in the interval where the sign of the function changes. The process continues until the interval becomes sufficiently small or an exact value of the root is found.

11. Create a singly linked list of integers (read numbers from a file); output the content of the list on the screen; find the largest number.

12. Create two sets, A and B, of the same size n, containing positive integers that do not intersect (arrays with different numbers). Find all pairs <a, b> from n input pairs, such that a belongs to A and is even, greater than 10, and b belongs to B and is odd and divisible by 5.

13. Develop a program that determines the number of n-digit binary numbers that do not contain consecutive ones. (Hint: The number starts with either zero or one. If it starts with zero, the number of options is determined by the remaining *(n-1)* digits. If it starts with one, what should be the next digit?)

14. Given a text (a string of characters). Reverse the sequence of characters that starts from index *start* and ends at index *end*.

15. Enter an array of n positive (>0) integers. Determine if it is possible to select numbers from this array whose sum equals the number s. If a variant exists, output the result on the screen. All numbers are entered from the keyboard.

For example, input data: 7, 5, 4, 4, 1; s = 10; 10 = 5 + 4 + 1.

16. Enter two arrays, such that their elements are sorted in ascending order. Develop a program to merge the two arrays into one array, sorted in ascending order.

17. Determine the value of number *a* raised to the power of *n*. Enter the values of *a* and *n* from the keyboard.

18. Compute the greatest common divisor (GCD) of numbers *a* and *b*

using the Euclidean algorithm. Enter the values of *a* and *b* from the keyboard.

19. Factorize a natural number a (integer type) into prime factors.
20. For given x and n, determine the value of the function.

$$P(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + .. + \frac{x^{2n+1}}{(2n+1)!}.$$

Checkpoint questions

1. Which object is called recursive?
2. What should you pay special attention to developing a recursive algorithm?
3. Which function is called directly recursive? Give an example.
4. Which function is called indirectly recursive? Give an example.
5. Give examples of recursive definition of functions.
6. What is the power of recursive definition?
7. What is a "recursive call tree"? Give examples.
8. Why do recursive programs need more memory than iterative programs?
9. What is written to the stack when a function is called?
10. What happens when the called function finishes its work?
11. Write the iterative version of this recursive function

```
int  FR (int n)
{    if (n==0) return 1;
      return n* F(n-1);
}
```

12. Write down the recursive version of this iterative function

```
int  FI (int key)
{   for (long i=0; i<N; i++)
         if( m[i]== key) return i;
    return -1;
         }
```

PRACTICAL ASSIGNMENT 3. INTERNAL REPRESENTATION OF BASIC DATA STRUCTURES

Goal: acquire and consolidate knowledge about the internal (computer) representation of numerical data types in programming languages.

Preparatory work:

- information and its representation in a computer;
- memory addressing;
- numeric data types: integers and floating-point numbers;
- errors in presenting floating-point numbers;
- static data structures - arrays.

General Information. Software development is a creative process, but there are many standard methods, techniques, and algorithms that simplify the programming process. Knowledge of such algorithms constitutes the subject of programming techniques. A prerequisite for mastering these techniques is a deep understanding of data structures because data is the material on which algorithms are based. The organization of data handling impacts the efficiency of algorithms, and in some cases, their correctness. Therefore, it is crucial to be able to choose data structures that correspond to the given task, estimate the required memory space, and evaluate the execution time of the future program.

The first thing encountered in programming is numerical data.

Integers: Data of integer type is stored in the computer's memory using the two's complement representation. For positive numbers, the two's complement is the same as the direct representation. The most significant bit of the most significant byte in memory, allocated for storing integers, is considered the sign bit.

Example 1: Representation of the integer number 258 in memory using the int format.

In the binary numeral system, this number can be represented as follows:

$$258_{10} = 256_{10} + 2_{10} = 2^8{}_{10} + 2^1{}_{10} = 1\ 00000010.$$

Integer data is stored in two or four bytes. The four-byte representation of the number, padded with leading zeros will be as follows:

Low byte 00000010 00000001 00000000 **0**0000000 High byte

The sign bit (indicated in bold) is equal to zero, which signifies positive numbers. Therefore, the number 258 is positive.

Example 2: Representation of the integer number 258 in memory using the int format. Let's write the direct, reverse, and two's complement representations of this number in the four-byte format.

Direct code: 00000000 00000000 00000001 00000010

Reverce code: 11111111 11111111 11111110 11111101

The two`s comlement: 11111111 11111111 11111110 11111110

Internal Representation:

(Low byte) 11111110 11111110 11111111 **1**1111111 (High byte).

The sign bit (indicated in bold) is equal to one, which indicates negative numbers; the number -258 is negative.

Real numbers. Data of real types are represented in memory as the following components: sign of the mantissa, normalized mantissa, and exponent. All components are represented in straight code.

The exponent (E) is determined as follows: $E = 2^{n-1} + p - 1$,

where p is the order of the number, determined during its normalization; n is the number of bits allocated for the exponent, defined by the data type.

The normalized mantissa (M) belongs to the interval: $1 \leq M < 2$. The integer part of the normalized mantissa is usually not stored in memory; this is called the hidden one. The sign bit in high byte of memory allocated for storing real numbers indicates the sign.

Example 3. Representation of the number 10.25 in memory in the float format.

The integer and fractional parts of the number are converted separately into binary notation.

Integer part: $10_{10} = 1010_2$. Fractional part 0,25 ×2

→ 0,50 ×2

→ 1,00 ×2

The arrow indicates the values of fractional binary digits: 0,01.

Thus, the entire number before normalization is: 1010,01

After normalization, the number becomes: 1,01001

The comma is moved three positions forward during normalization, so the order p = 3.

The memory size for data of type float is 4 B, and the exponent has 8 bits. Therefore, $E = 2^{8-1} + 3 - 1 = 2^7 + 2 = 2^7 + 2^1$.

In binary representation, the exponent will be: 10000010.

The internal representation of the number in memory is:

Low byte 00000000 00000000 00100100 **0**1000001 High byte

The mantissa is stored in memory without the hidden one.

To check their knowledge when studying this material, students can use a specially designed educational program. It describes all the bits of the number with labels indicating their purpose.

Example 4. Representation of the number 15.18 in memory in the double format.

The integer and fractional parts of the number are converted separately into binary notation.

Integer part $15_{10}=1111_2$. Fractional part 0,75 ×2

→ 1,50 ×2

→ 1,00 ×2

The arrow indicates the values of fractional binary digits 0,11.

Thus, the entire number before normalization is: 1111,11

After normalization, the number becomes: 1,11111

The comma is moved three positions forward during normalization, so the order p = 3.

The memory size for data of type double is 8 B, and the exponent has 11 bits.

Therefore, $E = 2^{n-1} + 3 - 1 = 2^{10} + 2 = 2^{10} + 2^1$ (1000000 0010)

The internal representation of the number in memory is:

(Low byte) 00000000 00000000 00000000 00000000

00000000 10000000 00101111 11000000 (High byte).

The mantissa is stored without the hidden one.

Arrays. Often in programming, the same algorithm is designed to process not just one element but a set of elements. A set of data of the same type is called an array, which is characterized by:

– a fixed set of elements of the same type;

– each element has a unique set of index values;

– the number of indices determines the dimensionality of the array, for example, two indices indicate a two-dimensional array, three indices represent a three-dimensional array, and one index indicates a one-dimensional array or vector;

– accessing an element of the array is done using the array's name and the values of its indices for that element.

In the C/C++ programming language, indexing of array elements always starts from zero. The elements of an array can have complex structures. The most important operation for an array is accessing a specific element. For this, it is necessary to calculate the address of the element. The compiler replaces each access to an array element with an expression that computes the address of that

specific element. Thus, accessing an element of a vector will be replaced by the following expression:

$$@Name[i] = @Name + i \cdot Sizeof(type).$$

Typical tasks

Task 1. Display the unsigned binary representation of integer type data on the monitor.

Program code

```
#include <iostream>
#include <conio.h>
using namespace std;

void BYTE(unsigned char A) // outputting the contents of the byte
{        for(int bit = 128; bit >= 1; bit >>= 1)
                 cout << (A & bit ? '1' : '0');
         cout << ' ';
}
void main (void)
 {   unsigned x;
     cout<<"Enter unsigned int value   " ;
     cin >> x;
     unsigned char *p =  (unsigned char*) &x;
     for(int byte = 0; byte < sizeof(unsigned); byte++, p++ )
         BYTE(*p);
     cout << endl;
}
```

The result of the program:

```
Enter unsigned int value   258
```

00000010 00000001 00000000 00000000

Task 2. Display a binary representation of data of integer type *int* and valid type *double* on the monitor. Use the template function.

```
#include "stdlib.h"
#include "iostream"
#define PB(value)(byte*)(value)
using namespace std;
typedef unsigned char byte;
typedef unsigned int dword;
template <class T> void show (const T &value)
{        for (dword i=0; i < sizeof(T);i++)
         {        byte Byte =*(PB(&value)+i);
                  for (dword j=0; j<8; j++)
                           cout << dword((Byte >> (7-j))&1);
                  cout << ' ';
         }
         cout << endl;
```

```
}
void main()
{        int val1=0;
double val2=0;
         cout<<"Enter int value :";
         cin>>val1;
         show(val1);
cout<<"Enter double value :";
         cin>>val2;
         show(val2);
}
```

The result of the program:

Enter int value : 12

00001100 00000000 00000000 00000000

Enter double value : 12.34

10101110 01000111 11100001 01111010 00010100 10101110 00101000 01000000

Individual tasks

Write a program that displays the internal (computer) data representation of four types on the screen. Select the data types according to the table. 3.1 according to your number in the group journal. Choose the type of array elements at your discretion.

Based on the results of the work, prepare a report on the practical assignment, where the obtained results and explanations are given, and conclusions are drawn.

Table 3.1 – Individual tasks

№ З/П	integer	short int	long int	float	double	long double	char	[] ***n***-вимірний
1	2	3	4	5	6	7	8	9
1	*			*			*	1
2	*			*			*	2
3	*			*			*	3

4		*			*		*	1
5		*			*		*	2
6		*			*		*	1
7			*			*	*	2
8			*			*	*	3
9			*			*	*	1
10	*				*		*	2
11	*				*		*	3
12	*				*		*	1
13		*		*			*	2
14		*		*			*	3
15		*		*			*	1
16			*		*		*	2
17			*		*		*	3

1	2	3	4	5	6	7	8	9
18			*		*		*	1
19	*					*	*	2
20	*					*	*	3
21	*					*	*	1
22		*				*	*	2
23		*				*	*	3
24		*				*	*	1

Note: the data types in the C/C++ language are presented in the table.

Checkpoint questions

1. In the form of what components are data of integer types presented in memory?

2. In what code does integer data appear in memory?

3. How much memory is allocated for data type short integer, integer, long integer?

4. In the form of which components are data of real types presented in memory?

5. In what code is data of real types presented in memory?

6. How are the numbers 255 and -255 represented in memory in short integer format?

7. How are the numbers –2 and 2 represented in memory in short real format?

8. What is determined by the error of presentation of real numbers?

9. What is a "hidden unit"? Why is it needed?

10. How accurate are real numbers?

11. What integer is stored in memory (in the decimal number system), if its physical representation (memory content) in the short int format is: 11111110 ?

12. What real number is stored in memory (in the decimal number system) if its physical representation (memory content) in float format is:

(Low byte) 0..0 0..0 11000000 11000000 (High byte)

PRACTICAL ASSIGNMENT 4. INTERNAL REPRESENTATION OF INTEGRATED DATA STRUCTURES

Goal: to acquire and consolidate knowledge about the internal representation of integrated data structures in programming languages.

Preparatory work:

- physical and logical presentation of data;
- integrated data types;
- data alignment.

General Information.

Structure is a finite set of fields of different data types, united by a common name. Structures are an extremely convenient means of representing program models of real objects in a subject area, as each such object often has a set of properties characterized by data of different types.

The C++ compiler represents a structure in memory as a sequence of fields occupying a contiguous block of memory. With this organization, the address of a field entry is determined by the value of the pointer to the beginning of the allocated memory block and the offset of the field relative to the start. This physical representation of structures provides memory efficiency but requires time to compute the addresses of the structure fields.

A structure may have a variant part. In this case, the structure consists of two parts. The first part describes fields that are common to all groups of objects modeled by the structure. Among these fields, there is usually one field whose value allows identifying the group to which this object belongs and, therefore, which variant from the second part of the structure should be used for processing. The second part of the structure contains descriptions of unique properties that do not overlap - a separate description is provided for each subset of such properties. C++ language rules require naming each variant. When accessing a field in the variant part of the record, its identification is complicated, and you need to specify the following:

<variable-record name>.<variant name>.<field name>

For a structure with variants, a memory area is allocated sufficient to accommodate the largest variant. If the allocated memory is used for a smaller variant, part of it remains unused. The part of the record common to all variants is located in such a way that the offsets of all fields relative to the start of the record are the same for all variants. Obviously, the simplest way to achieve this is by placing common fields at the beginning of the structure, but it is not strictly necessary. The variant part can be placed between the fields of the common part. Since in each case, the variant part has a fixed maximum size, the offsets of the common part fields will also remain fixed.

Data alignment is a very important concept that every programmer working with memory directly should take into account. Data alignment affects program execution speed and sometimes determines whether they will work at all. For example, most 32-bit computers are designed in such a way that machine words are read quickly when they are combined in quads (four bytes each) and addressed from zero, while machine words with addresses not divisible by 4 are read slowly. Therefore, most programming language compilers (including C) perform an optimization in which padding bytes are inserted between the fields of a structure to ensure that all fields are aligned.

For example, for the structure:

```
struct
{   char x; // 1 b
    int y;  // 4 b
    char z; // 1 b
};
```

12 bytes of memory will be allocated as shown in Figure 4.1.

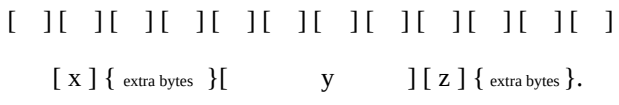


Figure 4.1 – Example of alignment of structure fields

As a result, an additional 6 bytes of memory will be allocated, which are not used in any way.

Data alignment means that the address, which is aligned, is determined as a numeric address modulo a power of 2. A data element is called naturally aligned if its address is aligned to the size of that element. If not, it will be unaligned. For example, an 8-byte floating-point data element is naturally aligned if the address used for its identification is aligned to 8. Programming language compilers attempt to distribute data in such a way as to avoid data inconsistencies.

The process of data alignment and, consequently, the size of a structure in memory, depends on compiler settings and directives in the program code. In most cases, there is no need to worry about alignment because the default alignment is already optimal.

In the following code example, it demonstrates how the compiler arranges the padded structure in memory:

```
struct Tstr2
{  char a;     // 1 byte
   char _a[7];  // addition to 8 byte
   double b;   // 8 bytes
   short c;    // 2 bytes
   char _3[6];  // addition to 8 byte
 };
```

When using an array of aligned data structures, only the first structure in the array is guaranteed to be aligned. As a result, the compiler adds additional unused bytes at the end of each processed structure. This process ensures that the size of the structure is a multiple of 2, 4, or 8..

Sometimes, significant performance gains or efficient memory usage can be achieved by adjusting the alignment for data structures.

Customizing alignment using compiler directives. Before Visual Studio 2015, keywords specific to Microsoft systems were used to indicate alignment exceeding the default value. These keywords include***_alignof(<type>)and __declspec(align(<number>))*** (with double underscores),

_alignof(<type>) returns the current alignment requirements for the given type,

__declspec(align(<number>)) is used for forced alignment based on the data type (provided that it is greater or equal to what ***_alignof*** would specify for that data type).

In the following example, the usage of the ***align*** and ***alignof*** keywords is shown:

```
__declspec(align(32)) struct Str1 // alignment is set to a limit of 32 B
{  int a, b, c, d, e;
};
void main ()
{   Str1 str;
    std::cout << sizeof (str) << "  " << _alignof (str) << std::endl;
}
```

The compiler directive #pragma pack (push, <number>) sets the alignment to <number> bytes.

The #pragma pack(pop) directive returns to the previous alignment. It is recommended to restore the previous alignment to avoid runtime errors. It is written after the structure description.

A typical task

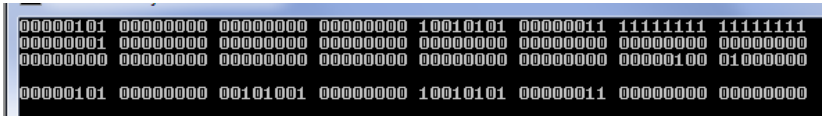
Displaying the physical representation of a data type "structure" with bit fields and variant parts on the monitor.

Program code

```
#include <iostream>
#include <conio.h>
using namespace std;

void BYTE(unsigned char A) // outputting the contents of the byte
{        for(int bit = 128; bit >= 1; bit >>= 1)
                 cout << (A & bit ? '1' : '0');
         cout << ' ';
}
struct Tstruct1
{     int s;       /* 4 bytes */
short flip1:1;
short flip2:1;
short flip3:1;
short flip4:1;
short flip5:1;
short flip6:4;
     short flip7:4;
     union
     {  struct
         {   bool  b;
             double dl;
         } prim1;
       struct
         {  short st;
```

```
        }       prim2;
    } un;
};
struct Tstruct2
{    short s;     /* 4 bytes */
     int flip1:1;
     int flip2:1;
     int flip3:1;
     int flip4:1;
     int flip5:1;
     int flip6:4;
     int flip7:4;
};
void main()
{    Tstruct1 obj1 = {5, 1, 0, 1, 0, 1, 12, 1};
     obj1.un.prim1.dl = 2.5;
     obj1.un.prim1.b = true;
     unsigned char *p =  (unsigned char*) &obj1;
     int byte = 0;
     for( ; byte < sizeof(Tstruct1); byte++, p++ )
     {   if (byte && !( byte % 8) ) cout << endl;
                 BYTE(*p);
     }
     cout << endl;
     cout << endl;
     Tstruct2 obj2 = {5, 1, 0, 1, 0, 1, 12, 1};
     p =  (unsigned char*) &obj2;
     for(int byte = 0; byte < sizeof(Tstruct2); byte++, p++ )
         BYTE(*p);
     cout << endl;  cout << endl;
}
```

The result of the program:



Conclusions based on the obtained results

The obtained results indicate that 24 B of memory is allocated to the variable *obj1* so that:

- for field s – 4 B;
- for fields *flip1* - *flip7* – 2 B, another 2 B added for alignment;
- for the variant part – 16 B, 7 B are added after the one-byte field *b* .

The variable *obj2* is allocated 8 B of memory so that:

– for field *s* – 2 B, another 2 B added for alignment;

– for fields *flip1* - *flip7* – 4 B.

Individual tasks

Write a program that displays the internal representation of a structure with a variant part and bit fields, as well as an array of structures. List of properties and corresponding types of fields for objects from the table. 4.1 choose at your own discretion.

Explore how alignment of data and structure fields is performed.

Compare data access times with and without alignment. Based on the results of the work, prepare a report on the practical assignment, where the obtained results and explanations are given, and conclusions are drawn.

Table 4.1 – Individual tasks

N	Object	Object type	State (yes/no)
1	Automobile transport	passenger, cargo	– there is cabin heating, – headlights are off
2	Education	higher, secondary	- technical, - there is employment
3	Application program	office, accounting	- platform dependent, - paid
4	Railway train	passenger, cargo	- traffic according to the schedule, - stops at all stations
5	Game	computer, board	- for children, - a collective game

6	Sea transport	dry cargo, tanker	– loaded, - foreign
7	A data set is an array	static, dynamic	– reading from a file, – sorted
8	Animals	domestic, wild	- even-toed, - herbivores
9	Power source	battery, accumulator	– charged, – reusable
10	Educational level	master's degree, bachelor's degree	- admission by exams, - defense of the graduation thesis
11	Geometric figures	Three-dimensional, two-dimensional	- multi-colored, - paper
12	Phone	landline, mobile	- button, - has access to the Internet
13	Trees	garden, wild	- stone - early ripe
14	Enterprise	state, private	- presence of a website, - presence of corporate culture
15	Coffee	coffee beans, instant coffee	– the presence of caffeine, - ease of preparation
16	Furniture	office, for home	– cabinet furniture, - free modification

17	Medicine	anti-inflammatory, antipyretic	– by prescription - domestic production
18	literature	technical, fiction	– colorful drawings, - hardcover

Checkpoint questions

1. How are Little Endian machines different from Big Endian machines?
2. What is "alignment" and why is it necessary?
3. How can data alignment be changed?
4. What can the physical representation of the structure be?
5. How is memory allocated for structures with variant parts?
6. How is memory allocated for structures with bit fields?
7. Give an example of accessing the fields of the structure from the variant part.
8. What is the purpose of the structure descriptor, who creates it?
9. Does the amount of memory allocated for a structure depend on the sequence of fields in its composition?
10. What compiler directives are intended to affect data alignment?
11. Does the address that the compiler assigns to data depend on the data type?
12. What determines the size of a structure in computer memory?
13. How are structures aligned in a structure array?

PRACTICAL ASSIGNMENT 5. PHYSICAL PRESENTATION OF SPECIFIC ARRAYS

Goal: acquisition and consolidation of programming skills of placing specific arrays in memory.

Preparatory work:

- physical and logical presentation of arrays;
- concept of descriptor;
- specific arrays: sparse, associative, symmetric, segment tree.

General Information

A sparse array, or a sparse matrix, is an array in which not all elements are used, available, or needed at the current moment. A sparse matrix is a matrix with mostly zero or background elements. In the case when most of the elements of the matrix are non-zero, the matrix is considered dense. There is no consensus among experts in determining exactly what number of non-zero elements makes a matrix sparse. Different authors offer different options.

Sparse arrays are useful under the following conditions:

1) The size of the array required by the application is quite large, possibly exceeding the available memory capacity;

2) Not all elements of the array are informative and used.

Thus, a sparse array is, as a rule, a large but sparsely filled array, most of whose elements are not informative (their default values are 0 or null, and may be other, but must be the same). Storing a large number of zeros in an array is inefficient for both storage and array processing. In a sparse array, access to background elements is also possible. In this case, the array will return null (if it is an array of numbers) or null (in the case of an array of objects).

If you need an array of a larger size than the computer's capabilities allow, it must be implemented in some other way. Even if a large array is located in memory, its creation can significantly reduce available system resources, because the memory occupied by the large array is unavailable to the rest of the program and other processes running on the system. And this can negatively affect the overall performance of the program or the computer as a whole. In

cases where not all elements of the array will be used, allocating memory for the entire array is a waste of system resources.

To address the issues caused by the memory requirements of large sparsely populated arrays, unusual algorithms for working with sparse arrays have been proposed. All of them are characterized by one common feature: memory for array elements is allocated only when necessary. Therefore, the advantage of a sparse array is that its storage requires exactly as much memory as is needed to store only those elements that are actually used. At the same time, free memory can be used for other purposes. In addition, such methods allow you to create very large arrays, the size of which is much larger than the size of ordinary arrays that the system allows.

A typical task

Develop a method of memory-efficient storage of a matrix of integers in which the zero elements are located below the main diagonal.

In the internal representation of the array, there is no need to store null (background) elements - they are those for which the condition is fulfilled:

$$M[x, y] = 0 \text{ if } x < y.$$

If zero elements are excluded from storage and the matrix is presented in the form of a one-dimensional array, then the linearization formula (transition from two-coordinate to one-coordinate representation) is written as:

$$j = \sum_{i=0}^{x}(N - i) + (y - x),$$

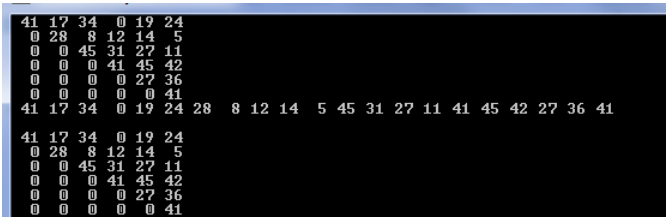
where x, y are column and row numbers, accordingly, in the sparse matrix; N is the number of elements in the line; j is the number of the corresponding element in the one-dimensional array.

Program code

```
#define N 6
int NewIndex(int x, int y)     // index listing
{   int j=0;
    for(int i=0; i<x; i++) j+= N-i;
    return j+y-x;
}
void Put(int vec [], int x, int y, int v)  // Writing to a vector (compression)
{  if (y >=x) vec[NewIndex(x, y)] = v;
}
int Get(int vec [], int x, int y) // reading from a vector
{  if (y >=x) return vec[NewIndex(x, y)];
   else return 0;
```

```
}
void RandArray(int a[N][N]) // formation of the initial array
{  for(int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      if (j >=i) a[i][j] = rand()%50;
      else a[i][j] = 0;
}
void PrintArray(int a[N][N])
{    for(int i = 0; i < N; i++)
     {  for (int j = 0; j < N; j++)
           printf("%3i", a[i][j]);
        printf("\n");
     }
}
void main()
{  int vec[N*N/2+N/2];
   int array[N][N];
   RandArray(array);
   PrintArray(array);
   for(int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
         Put(vec, i, j, array[i][j]);          // array compression
   for(int i = 0; i < N*N/2+N/2; i++)          // output of the compression result
         printf("%3i", vec[i]);
    printf("\n\n");
    for(int i = 0; i < N; i++)
    { for (int j = 0; j < N; j++)    // reading from a compressed view
            printf("%3i", Get(vec,i,j));
      printf("\n");
     }
}
```

The result of the program:



Note: the given program does not perform testing of timed access to the elements of the array with its complete and economical (concise) representation in memory.

Individual tasks

Develop a method for efficient storage of a given sparse table in memory, where integer numbers are recorded. Develop functions that provide access to table elements by row and column numbers.

In the program, ensure the writing and reading of all table elements.

Determine and compare the access time to table elements in traditional and efficient memory representations. Draw conclusions.

Choose a task from table 5.1 according to your group journal number.

Table 5.1 – Variants of individual tasks

N	The content of the sparse matrix
1	Zero elements are located in the left half of the matrix
2	Zero elements are located in the right half of the matrix
3	Zero elements are located in the upper half of the matrix
4	Zero elements are located in the lower half of the matrix
5	All elements of odd rows are zero
6	All elements of even columns are zero
7	Zero elements are arranged in staggered order, starting from the 0th element of the 0th row
8	The zero elements are arranged in staggered order, starting from the 1st element of the 0th row
9	Zero elements are located in places with even row and column indices
10	The zero elements are located at positions with odd indices of rows and columns.
11	Zero elements are located above the main diagonal on odd lines and below the main diagonal - on even lines

12	Zero elements are located in the first and third thirds of the rows of the matrix
13	Zero elements are located in the left and right thirds of the matrix columns
14	Zero elements are located in the first and third thirds of the rows and in the left and right thirds of the columns of the matrix
15	Zero elements are located on the main diagonal and in the upper half of the matrix above the diagonal
16	Zero elements are located on the main diagonal and in the lower half of the matrix below the diagonal
17	Zero elements are located in the left and right quarters of the matrix
18	Zero elements are located in the upper and lower quarters of the matrix
19	Zero elements are located on rows whose indices are multiples of three.
20	Zero elements are located in the columns whose indices are multiples of three.
21	Zero elements are located in the upper third of the rows and the middle third of the columns.
22	Zero elements are located in the upper third of the rows and the first and third thirds of the columns.
23	The zero elements are located in the upper and lower triangles, assuming the matrix is divided into four triangles by diagonals.
24	The zero elements are located in the left and right triangles, under the conditions of dividing the matrix diagonally into 4 triangles

Checkpoint questions

1. What is the "descriptor" of the array, what is its purpose?

2. How are arrays represented in memory?
3. How to determine the amount of memory needed to write an array?
4. How is the address of an array element determined?
5. Does the access time to an element of an array depend on the dimension of this array and why?
6. How do dynamic arrays differ from static arrays?
7. What are the characteristics of arrays of variable length, how do they differ from dynamic ones?
8. What are "heterogeneous arrays"?
9. At the logical level, how are associative arrays presented?
10. What arrays are considered symmetric, how they are presented at the physical level?
11. Under what conditions are arrays considered sparse?
12. What is the purpose of segment trees and how do they relate to arrays?
13. What is the V-list, what are its advantages?
14. When are parallel arrays needed?
15. What is the purpose of hash tables, how are they created?

PRACTICAL ASSIGNMENT 6. REPRESENTATION OF STRINGS IN MEMORY

Goal: Acquire practical skills and reinforce knowledge about possible string representations and operations on strings.

Preparatory work:

- String representations.
- String operations.
- String processing tools in programming languages.
- Text processors.

General Information

A string is a linearly ordered sequence of characters belonging to a finite set of characters known as an alphabet. Strings have the following important properties:

- their length usually varies, while the alphabet remains fixed;
- access to the characters in a string can be done from either end of the sequence, meaning the order of the sequence is essential, not its indexing. Therefore, strings are also referred to as chains;
- the main purpose of accessing a string is often not to retrieve individual elements (although it is not excluded), but rather to access a certain substring within the string.

When discussing strings, text strings are often implied – strings composed of characters from the alphabet of any chosen language, digits, punctuation marks, and other special characters. Text strings represent the most universal form of information representation. Depending on specific tasks, the mutability of strings can range from complete absence to practically unlimited possibilities of change. The degree of mutability of strings determines their physical representation in memory and the characteristics of operations performed on

them. In most programming languages, strings are presented as semi-static structures.

Operations on strings. The basic operations on strings are as follows:

- determining the length of a string; string assignment;
- concatenation (joining) of strings;
- substring extraction;
- searching for substring occurrences.

The operation of *determining the length of a string* is a function that returns an integer, which represents the current number of characters in the string.

The *assignment* operation has the same meaning as for other data types.

String comparison is performed according to the following rules: the first characters of two strings are compared. If the characters are not equal, the string that contains the character with a position closer to the beginning of the alphabet is considered smaller. If the characters are equal, the second, third, etc., characters are compared. When reaching the end of one of the strings, the string with the smaller length is considered smaller. When the lengths of the strings are equal, and all characters are pairwise equal, the strings are considered equal.

The result of *concatenating two strings* is a new string whose length is equal to the sum of the lengths of the operand strings, and the value corresponds to the value of the first operand, followed immediately by the value of the second operand. The concatenation operation produces a result whose length is generally greater than the lengths of the operands.

As in all string operations that can increase the length of the string (assignment, concatenation, complex operations), there is a possibility that the length of the result will exceed the allocated memory size. Naturally, this problem arises only in languages where the length of a string is limited. There are three possible ways to solve this problem, determined by the rules of the language or compilation modes:

– not controlling such an overflow; encountering such a situation inevitably leads to an error that is difficult to localize during program execution;

– terminating the program abruptly with error localization and diagnostics;

– limiting the length of the result according to the allocated memory size.

The *substring extraction operation* selects a sequence of characters from the initial string, starting from a given position ***n*** and having a specified length ***l***. When implementing the substring extraction operation, the function must define a rule for obtaining the result in the case when the initial position ***n*** is such that the remaining part of the initial string has a length less than the specified length ***l***, or even when n exceeds the length of the initial string. Possible options for such a rule include:

– abruptly terminating the program with error diagnostics;

– forming a result of a smaller length than the specified one, possibly even an empty string.

The *substring search* operation finds the position of the first occurrence of a given substring pattern in the target string. The result of the operation can be the position number in the original string where the occurrence of the pattern begins, or a pointer to the start of the occurrence. If the substring pattern is not found, the result of the operation should be a special value, such as the zero position number or a null pointer.

Based on the basic string operations, any other, even complex, operations on strings can be implemented. For example, the operation of removing characters with numbers from **n1** to **n2**, inclusive, from a string can be implemented as a sequence of such steps:

– extracting a substring from the given string, starting from position **0**, of length ***(n1-1)*** characters;

– extracting a substring from the original string, starting from position ***(n2+1)***, of length equal to the length of the given string minus ***n2***;

– concatenating the substrings obtained in the previous steps.

However, with the aim of increasing efficiency, some secondary operations can also be implemented as basic ones, using their algorithms with direct access

to the physical structure of the string.

The representation of strings in memory depends on how variable the strings are in each specific task, and the means of representation vary from completely static to dynamic. String vector representation (vector of fixed length, vector with a counter, vector with an end marker, vector with a controlled length) and list representation (character-linked or block-linked with blocks of fixed, variable, and controlled length) are distinguished. Lists can be single-linked or double-linked. Universal programming languages mostly provide for working with strings of variable length, but the maximum length of a string must be specified when it is created.

A typical task

Implement and verify the following operations on strings:

– Determine the position of the first occurrence of character ***c*** in the string ***s***. If c is found, return the index of character ***c*** in ***s***. Otherwise, return -1.

– Determine and return the length of the initial substring of string ***s2*** that consists of characters present in string ***s1***.

Program code

```
#include <iostream>
#include <cstring>

// Determines the occurrence of a character in a string

int CharInString(const char s[], char c)
{
    int i = 0;
    for (const char *p = s; *p; p++, i++)
        if (*p == c)
            return i;
    return -1;
}

// Searches for the occurrence of a substring in a string

int SubstringInString(const char s[], const char sb[], int &l)
{
    l = 0;
    int i = 0, j = 0;
    while (s[i] && sb[j] && s[i] != sb[j]) // finding the first match of the first
character of the substring
        i++;
    if (s[i] == sb[j])
    {
```

```
        while (s[i] && sb[j] && s[i] == sb[j]) // determining the length of the
string of matched characters
            j++, i++;
        l = j;
        return i - l;
    }
    return -1;
}

int main(void)
{
    setlocale(LC_ALL, "Rus");
    char *str = new char[50];
    char *substr = new char[50];
    int poz;
    char simbol;
    std::cout << "Which string? : ";
    std::cin.getline(str, 50);
    std::cout << "Which substring? : ";
    std::cin.getline(substr, 50);
    std::cout << "Which character? : ";
    std::cin >> simbol;
    poz = CharInString(str, simbol);
    if (poz >= 0)
        std::cout << poz << std::endl;
    else
        std::cout << "Character not found\n";
    int len;
    poz = SubstringInString(str, substr, len);
    if (poz >= 0)
        std::cout << poz << " len = " << len << std::endl;
    else
        std::cout << "Substring not found in string\n";
    delete[] str;
    delete[] substr;
    return 0;
}
```

The result of the program:

```
Which string? : This is test example
Which substring? : tes
Which character? : m
16
8 len = 3
```

Individual tasks

Write a program that predicts the execution of the indicated operation on strings, provided that strings are represented in memory in two ways. Compare

the representation of lines by the specified methods according to the following indicators:

- amount of used memory;
- function execution time.

Select the operation and methods of rendering lines from table 6.1.

Table 6.1 – Variants of individual tasks

Variant	1	2	3	4	5	6	7	8	9	10	11	12	13
Function	1	2	3	4	5	6	7	8	9	10	11	12	13
row representation	4 8	3 7	2 6	1 5	4 9	3 5	2 6	1 7	4 8	3 9	2 5	14 6	4 7

Continuation of the table 6.1.

Variant	14	15	16	17	18	19	20	21	22	23	24	25
Function	1	2	3	4	5	6	7	8	9	10	11	12
row representation	3 8	2 9	1 5	4 6	3 7	2 8	1 9	4 5	3 6	2 7	1 8	4 9

Note: choose the task according to your number in the group journal.

The method of row representation

1) A vector of constant length.

2) Variable-length vector with a string termination marker.

3) Variable-length vector with a counter.

4) Vector with controlled string length (with a descriptor).

5) Character-linked representation using a singly linked list.

6) Character-linked representation using a doubly linked list.

7) Block-based representation with a fixed length.

8) Block-based representation with a variable length.

9) Block-based representation with a controlled length.

Please note that in tasks 7-9, the purpose of the block is left to your discretion

Functions

1) Determine the number of characters in the string *s*.
2) Concatenate lines *s1* та *s2*.
3) Replace all lowercase letters with uppercase letters in line *s*, starting from position *n*.
4) Delete a substring of length *k* characters from the line *s*, starting from position *n*.
5) Copy *n* characters from the string *s*, starting from the *m*th character.
6) Insert substring *s1* into line *s*, starting at position *n*.
7) Rewrite the string s so that the characters in it are written in reverse order.
8) Find substring *s1* in string *s*.
9) Determine the number of words of length *k* characters in line *s*.
10) Compare lines *s1* та *s2*.

11) Delete leading, trailing and multiple blanks in line *s*.

12) Replace all *c1* characters in line s with *c2* characters.

13) Remove duplicate words in line *s*, provided that the same words are written next to each other.

Checkpoint questions

1. What is a "string", what are its properties?
2. What basic operations are defined over strings?
3. What algorithm is used to compare strings?
4. What is string "concatenation"?
5. What are the known ways of representing strings in memory?
6. What is the algorithm for removing part of a string?
7. Why is a string descriptor created?
8. What are the advantages and disadvantages of displaying rows as an array?
9. What are the advantages and disadvantages of representing strings as a singly linked list?
10. What are the advantages and disadvantages of representing strings as a doubly linked list?
11. What is the purpose of blocks in block-based representation of strings?

PRACTICAL ASSIGNMENT 7. QUEUING ALGORITHMS

Goal: To gain practical experience and reinforce knowledge of stack, queue, priority queue representation, and their service disciplines.

Preparatory work:

- Arrays and lists

- Unordered and priority queues

- Queue service disciplines

General Information

A queue is a data structure that stores elements and provides access to them only in a certain order determined by priority. Arrays or linear lists can be used to implement queues.

A deque (double-ended queue) is a sequential data structure where the insertion and removal of elements can be done from either end of the set. The priority is determined by the arrival time of requests. There are specific cases of deques: LIFO (last-in, first-out) or stack and FIFO (first-in, first-out) queue.

In priority queues, the priority is determined by one or more parameters. When working with priority queues, the focus is not on the elements themselves, but on the requests that are being enqueued. It is possible to organize priority queues with priority-based insertion of elements into the queue and priority-based removal of requests from the queue.

Operations on queues include inserting an element into the queue, removing an element from the queue for servicing according to priority, determining the size of the queue, and clearing the queue.

Typical tasks

Task 1. Implement the Dijkstra's algorithm for constructing the reverse Polish notation (RPN) for a given arithmetic expression using a stack.

Explanation of the program text: The input string is *In*, and the output string is *Out*. The stack is implemented using a singly linked list. The stack stores

operation signs and opening parentheses. Reading from the stack is performed in three cases:

1) The priority of the current operation in the *In* string is lower than the priority of the operation at the top of the stack.

2) The current symbol in the *In* string is a closing parenthesis.

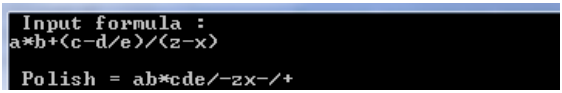
3) The *In* input string has ended.

Program code

```
struct Stack
{
   char c;          // Operation symbol in the stack
   Stack *Next;
};

int Prior(char);
Stack* Push(Stack*, char);
Stack* Pop(Stack*, char*);

int main()
{
   Stack *pointSt = nullptr;   // Stack of operations is empty, pointer = 0
   char a, In[N], Out[N];      // Input In and output Out strings
   int i = 0, j = 0;           // Current indexes for strings

   puts(" Input formula: ");
   gets(In);

   while (In[i]) // Analyzing characters in the input string In
   {
      if (In[i] == ')') // If the character is ')', pop from the stack to the
output string -> all operations
      {
         pointSt = Pop(pointSt, &a); // Remove an element from the stack
         while (a != '(') // Until the opening parenthesis is encountered
         {
            Out[j++] = a; // Remove an element from the stack and store it in the
string Out
            pointSt = Pop(pointSt, &a);
         }
      }
      else if (In[i] >= 'a' && In[i] <= 'z') // If the character is a letter, write
it to the string Out
      {
         Out[j++] = In[i];
      }
      else if (In[i] == '(') // If the character is an opening parenthesis, push it
to the stack
      {
         pointSt = Push(pointSt, In[i]);
      }
```

```
      else if (In[i] == '+' || In[i] == '-' || In[i] == '*' || In[i] == '/') // If
the character is an operation symbol, copy all operations with higher or equal
priority from the stack to the string Out
      {
         while (pointSt != nullptr && Prior(pointSt->c) >= Prior(In[i]))
         {
            pointSt = Pop(pointSt, &a); // Remove the next character
            Out[j++] = a; // Store it in the string Out
         }
         pointSt = Push(pointSt, In[i]); // Current character is pushed to the
stack
      }

      i++;
   } // End of the input string analysis loop

   // If the stack is not empty, copy all operations to the output string
   while (pointSt)
   {
      pointSt = Pop(pointSt, &a);
      Out[j++] = a;
   }

   Out[j] = '\0';

   printf("\n Polish = %s\n", Out); // Output the result on the screen
}

// Function for implementing operation priority
int Prior(char a)
{
   if (a == '*' || a == '/')
      return 3;
   else if (a == '+' || a == '-')
      return 2;
   else if (a == '(')
      return 1;
   else
      return 0;
}

// Add an element to the stack
Stack* Push(Stack* t, char s)
{
   Stack* t1 = (Stack*)malloc(sizeof(Stack));
   t1->c = s;
   t1->Next = t;
   return t1;
}

// Remove an element from the stack
Stack* Pop(Stack* t, char* s)
{
   Stack* t1 = t;
   *s = t->c;
   t = t->Next;
   free(t1);
   return t;
}
```

The result of the program:



Task 2. Develop a program for working with a priority queue. Ensure the servicing of requests with the maximum priority.

Explanation of the program text: The queue is implemented using an array. Enqueuing requests is done sequentially, at the end of the queue, while dequeuing is done based on priority. When configuring the program, commands and requests should be generated randomly. After each command, wait for any key press.

Program code

```
int och[512];            // queue with 512 requests
int point=0;              // queue end pointer

int AddEl(int a)          // adding a request to the queue
{
    if (point < 512)
        och[point++] = a;
    else
    {
        cout << "enough queue\n";      // queue is full
        return 0;
    }
    return 1;
}

void DelEl(void)          // reading a request from the queue
{
    if (point > 0)
    {
        int max = och[0], nmax = 0;     // finding the request with the maximum
parameter
        for (int i = 1; i < point; i++)
            if (max < och[i])
                max = och[i], nmax = i;
        cout << "read   " << max << endl;
        point--;            // shifting the queue
        for (int i = nmax; i < point; i++)
            och[i] = och[i+1];
    }
}
```

```
void PrintOch(void)      // outputting the content of the queue
{
    if (point == 0)
        cout << "queue empty \n";
    else
    {
        cout << "in queue:   ";
        for (int i = 0; i < point; i++)
            cout << och[i] << " ";
        cout << "\n";
    }
}

int main(void)
{
    int op, f;
    do
    {
        op = rand() % 2;        // 0 - write request, 1 - read
        cout << "enquiry " << op << "\n";
        if (op)
            DelEl();
        else
        {
            f = rand() % 15;        // request
            if (!AddEl(f))
                exit(1);
        }
        PrintOch();
        _getch();
    } while (1);
}
```

A fragment of the results of the program

```
enquiry 1
queue empty
enquiry 1
queue empty
enquiry 0
in queue:   10
enquiry 1
read   10
queue empty
enquiry 0
in queue:   3
enquiry 0
in queue:   3 7
enquiry 0
in queue:   3 7 5
enquiry 1
read   7
in queue:   3 5
```

Individual tasks

Develop functions that provide writing and reading operations from a priority queue, stack, or deque.

Use two data structures in each task to organize the specified data structure. Check the functionality of the developed functions. Randomly choose the sequence of write and read operations.

Compare the results and draw conclusions.

1. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using an array with shifting after each read operation and an array with shifting after reaching the memory boundary allocated for the queue. Priority: ***min*** value of a numeric parameter; in case of parameter equality - *LIFO*.

2. Deque: The deque is implemented using a cyclically filled array and a doubly linked list. Operations are performed from both ends of the deque.

3. *FIFO* Queue: The queue is implemented using an array with shifting after each read operation and an array with shifting after reaching the memory boundary allocated for the queue. The queue "grows" from a lower address to a higher address.

4. Priority Queue: Requests are enqueued sequentially to the end of the queue based on priority, and dequeued based on priority. The queue is implemented using an array and a linked list. Priority: ***min*** value of a numeric parameter; in case of parameter equality - *FIFO*.

5. Stack: The stack is implemented using a doubly linked list and an array, and it "grows" from a lower address to a higher address.

6. Deque: The deque is implemented using a cyclically filled array and a doubly linked list. Operations are performed from different ends of the deque.

7. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using a cyclically filled array ans array with shifting. Priority: ***max*** value of a numeric parameter; in case of parameter equality - *FIFO*.

8. *FIFO* Queue: The queue is implemented using an array with shifting after each read operation and a linked list. The queue "grows" from a higher

address to a lower address.

9. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the higher addresses (end of the queue). The queue is implemented using an array and a linked list. Priority: ***max*** value of a numeric parameter; in case of parameter equality - *FIFO*.

10. Deque: The deque is implemented using a cyclically filled array and shifting. Operations are performed from both ends of the deque.

11. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using a cyclically filled array and a linked list. Priority: ***max*** value of a numeric parameter; in case of parameter equality - *FIFO*.

12. FIFO Queue: The queue is implemented using a linked list and a cyclically filled array. The queue "grows" from a higher address to a lower address.

13. Deque: The deque is implemented using a cyclically filled array and shifting. Operations are performed from different ends of the deque.

14. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using an array with shifting after each read operation and an array with shifting after reaching the memory boundary allocated for the queue. Priority: ***max*** value of a numeric parameter; in case of parameter equality - *FIFO*.

15. Stack: The stack is implemented using a singly linked list and an array, and it "grows" from a higher address to a lower address.

16. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using a cyclically filled array and array with shifting. Priority: ***min*** value of a numeric parameter; in case of parameter equality - *LIFO*.

17. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the higher addresses (end of the queue). The queue is implemented using an array and a linked list. Priority: ***min*** value of a numeric

parameter; in case of parameter equality - *LIFO*.

18. FIFO Queue: The queue is implemented using a linked list and a cyclically filled array. The queue "grows" from a lower address to a higher address.

19. Priority Queue: Requests are enqueued based on priority, and dequeued sequentially from the lower addresses (start of the queue). The queue is implemented using a cyclically filled array and a linked list. Priority: ***min*** value of a numeric parameter; in case of parameter equality - *LIFO*.

20. Priority Queue: Requests are enqueued sequentially to the end of the queue, and dequeued based on priority. The queue is implemented using an array and a linked list. Priority: ***max*** value of a numeric parameter; in case of parameter equality - *FIFO*.

21. Stack: The stack is implemented using a singly linked list and an array, and it "grows" from a lower address to a higher address.

Checkpoint questions

1. Provide a definition of ‘queue’
2. What is a stack?
3. What are the known types of queues?
4. What data structures can be used to implement a stack?
5. What operations are typical for a queue?
6. What operations are typical for a stack?
7. What is the algorithm for reading requests from a stack?
8. What properties are inherent in a queue?
9. What is a drawback of a simple queue? How can it be addressed?
10. How does a circular queue differ from a simple queue?

11. What are the applications of queues and stacks?

12. How does an array-based stack differ from a linked list-based stack?

13. What is the difference between a write-priority queue and a read-priority queue?

14. How does a linked list-based stack differ from a regular linked list?

15. What is a deque?

16. What is the algorithm for removing a request from a priority queue where requests are placed sequentially as they arrive?

17. What is the algorithm for adding a request to a priority queue where requests are removed sequentially?

18. What mistake is made in the provided function for adding an element to a stack?

```
int StackPush(int elem)
{  St[--top]=elem;  return 1;
}
```

19. Provide an example of functions for adding an element to a deque.

20. Provide an example of functions for removing an element from a deque.

PRACTICAL ASSIGNMENT 8. LISTS

Goal: To acquire skills and reinforce knowledge while performing operations on linked lists and non-linear lists.

Preparatory work:

- data structures - lists: linear, circular, linked lists, multi-lists, non-linear lists;
- operations on lists.

General Information

A list is a collection of elements to which inclusion and exclusion operations can be applied. In memory, a list is represented as a set of descriptors and records with the same size and format, located arbitrarily in a certain memory region and linked together in a linearly ordered chain using pointers.

There are linear and non-linear lists. Linear lists can be further categorized as single-linked, double-linked, or circular lists.

Operations on linked linear lists are as follows:

- traversing the list: This involves sequentially accessing all elements in the list from its beginning to the end or until finding the desired element. In a circular list, the traversal should stop not based on the last element but after reaching the element from which the traversal started.
- insertion of an element at any position in the list;
- deleting an element from any position in the list;
- rearrangement of elements in the list.
- copying part of the list. A copy operation involves duplicating data in memory. During copying, the input list is preserved in memory, and a new list is created. However, the linkage fields in the new list are different since the elements of the new list are located at different addresses in memory.
- merging two lists: The merging operation involves combining two lists

into one. This operation is analogous to concatenating strings.

In all operations, the sequential correction of pointers is extremely important, which ensures the correct modification of some elements of the list without affecting others. If the correction of pointers is done incorrectly, part or even the entire list may be lost..

A non-linear multi-level list is a list whose elements can also be lists, as shown in Figure 8.1.

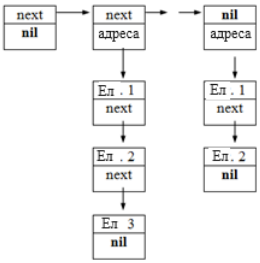


Figure 8.1 – List of lists

The elements of a non-linear list have more than one pointer within their structure. Non-linear branched lists are described by three characteristics: order, depth, and length.

Order describes the sequence in which elements appear inside the list.

Depth is the maximum level assigned to elements within the list or within any of its sublists. The depth is determined by the nesting of sublists within the list.

Length is the number of elements in the first level of the list.

The main operations on non-linear lists are:

– adding a node;

– removing a node;

– specific operation - traversing the list.

A binary tree is a non-linear list in which each element has a data field and two reference fields - left and right (Figure 8.2). It is characterized by the following properties:

- there is one node that is not referenced by any other node. This node is called the "root" or "parent node";
- there are nodes that do not reference to any other nodes - these are leaves.

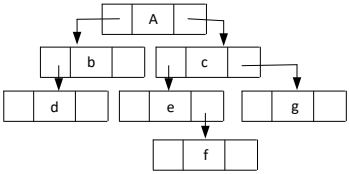


Figure 8.2 – Binary tree

The other nodes are branch nodes. The root and branch nodes refer to the left and right subtrees. Since a tree is essentially a recursive structure, all functions for working with trees are also recursive.

An orthogonal list (or multilist) is a structure, each element of which is included in more than one list at the same time and has the number of pointer fields corresponding to the number of lists. The implementation of each of the lists can be done as a singly or doubly linked non-cyclic or cyclic list. The technology for processing multislists does not differ from processing linear lists.

A typical task

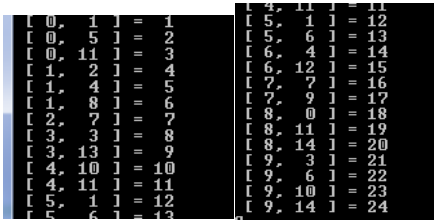
Develop a program to display a sparse matrix in the form of a multilist. Read data from a file. The value of the elements of the sparse matrix should be displayed in rows.

Program code

```
// Example of processing a sparse matrix - a procedure that prints the values of
matrix elements row by row.
#include <stdio.h>
#include <fstream>
#include <malloc.h>
#define NROW  10    // number of rows
#define NCOL  15    // number of columns

struct TMatrix    // structure definition for a node in the linked list
{
    int val;        // value of the element
    int row, col;   // row number, column number
    TMatrix* frow, * fcol;    // forward and down attributes for row and column
lists
};

void Read_File(TMatrix frow[], TMatrix fcol[])
{
    TMatrix* temp;
    TMatrix* p;
    FILE* file;
    file = fopen("Text1.txt", "r");
    while (!feof(file))
    {
        temp = (TMatrix*)malloc(sizeof(TMatrix));
        fscanf(file, "%i%i%i", &temp->row, &temp->col, &temp->val);
        p = &frow[temp->row];
        while (p->fcol)    // moving to the end of the row list
            p = p->fcol;    // move to the next column element
        temp->fcol = p->fcol;
        p->fcol = temp;
        p = &fcol[temp->col];
        while (p->frow)    // moving to the end of the column list
            p = p->frow;    // move to the next row element
        temp->frow = p->frow;
        p->frow = temp;
    }
    fclose(file);
}

void Init(TMatrix frow[], TMatrix fcol[])
{
    for (int i = 0; i < NROW; i++)
    {
        frow[i].fcol = 0;
        frow[i].frow = 0;
        frow[i].row = 0;
    }
    for (int i = 0; i < NCOL; i++)
    {
        fcol[i].fcol = 0;
        fcol[i].frow = 0;
        fcol[i].col = 0;
    }
}

void Print_Matrix(TMatrix frow[])
{
    TMatrix* p;    // auxiliary pointer for traversing
```

```
    for (int i = 0; i < NROW; i++)    // iterate through rows
    {
        p = frow[i].fcol;    // set the pointer to the first element of the row
list
        while (p)    // if the row list is not empty
        {
            printf(" [%2i,%3i ] = %2i\n", p->row, p->col, p->val);
            p = p->fcol;    // move to the next element in the row
        }
    }
}

void Delete_Matrix(TMatrix frow[])
{
    TMatrix* p, * pDel;    // auxiliary pointers for traversing the row
    for (int i = 0; i < NROW; i++)    // iterate through rows
    {
        p = frow[i].fcol;
        while (p)    // if the row list is not empty
        {
            pDel = p;
            p = p->fcol;    // move to the next element in the row
            delete pDel;
        }
    }
}

int main()
{
    TMatrix* pRow;    // array of pointers to the first nodes of the column lists
    pRow = (TMatrix*)malloc(NROW * sizeof(TMatrix));
    TMatrix* pCol;    // array of pointers to the first nodes of the row lists
    pCol = (TMatrix*)malloc(NCOL * sizeof(TMatrix));
    Init(pRow, pCol);
    Read_File(pRow, pCol);
    Print_Matrix(pRow);
    Delete_Matrix(pRow);
    delete pRow;
    delete pCol;
}
```

The result of the program:



Individual tasks

For variants of tasks 1 - 13, develop a program that creates a list of lists

(non-linear list). Provide the following functions:

– adding elements to the list and sublist (when adding an element to the main list, the corresponding sublist is also added);

– deleting items from the list and sublists (when deleting an item from the main list, the sublist associated with it is also deleted);

– outputting the contents of lists and sublists to the console;

– deleting lists.

Select the task from the table. 8.1 according to your number in the group journal.

Table 8.1 – Variants of individual tasks for creating non-linear lists

N	List	Sublist
1	Surnames of novels authors	Names of the author's novels
2	Name of the university	Names of university faculties
3	Indexes of educational groups	Surnames of students of the group
4	Names of countries	Cities of the country
5	Bus routes	Names of bus stops
6	Names of cities	Names of enterprises in the city
7	Shop names	List of goods for sale
8	Road numbers	Names of settlements along the road
9	Construction companies	List of surrendered objects
10	Departments of enterprises	Surnames of department employees
11	Names of specialties	Names of universities for the training of specialists in the specified specialties

12	Devices	Names of component nodes
13	Olympics	By years, the names of the winners

For the variants of tasks 14 - 20, develop a program that creates a multilist. Provide the following functions:

– adding elements to the list and sublists;

– deleting elements from the main list and sub-lists (when deleting an element from one of the lists, this element is not deleted from memory. And only when the element is not included in any of the lists, it is deleted from memory);

– outputting the contents of lists and sublists to the console;

– deleting lists.

Choose tasks for variants 14-20 from the table. 8.2 according to your number in the group journal. Choose the list of properties at your discretion.

Table 8.2 – Variants of individual tasks for creating multilists

N	Object	Lists
14	Institute students	1) all students of the institute; 2) foreign students; 3) students of the first year of study; 4) master's students.
15	Book fund	1) all books of the library; 2) technical literature; 3) fiction; 4) books for children
16	Residents of the	1) all residents of the building;

	apartment building	2) children; 3) working residents of the building; 4) pensioners.
17	Vehicles	1) all vehicles; 2) automotive; 3) railway; 4) marine
18	Languages	1) all programming languages; 2) high-level programming languages; 3) low-level programming languages; 4) modern programming languages
19	Goods	1) names of all goods; 2) goods of domestic production; 3) imported goods; 4) Chinese-made goods
20	Athletes	1) surnames of all athletes; 2) figure skaters; 3) hockey players; 4) gymnasts

Checkpoint questions

1. What are the differences between a list and an array?
2. What kinds of lists are there?
3. What operations are typical for linear lists?

4. What operations are typical for nonlinear lists?
5. What are the characteristics of non-linear branched lists?
6. What are the characteristics of a binary tree?
7. What is a multilist?
8. How does a circular list differ from a simple linear list?
9. What are the features of deleting elements from a multilist?
10. What are the features of deleting elements from a list of lists (non-linear list)?
11. How is the depth of a non-linear list determined?
12. What determines the length of a non-linear list?
13. What describes the order of a non-linear list?
14. What programming languages are designed exclusively for processing lists?
15. Construct a non-linear list for the arithmetic expression

$$(a+b)\cdot(c-d)/e$$

16. Construct a non-linear list for the arithmetic expression

$$d + f/(a+b)\cdot c$$

17. Name known methods for managing free dynamic memory.
18. Name algorithms for allocating free dynamic memory.
19. What are known methods for deallocating dynamic memory?
20. Give a description of the "best fit" algorithm for on-demand allocation of dynamic memory.
21. Give a description of the "first fit" algorithm for on-demand allocation of dynamic memory.

22. What are "holes" in memory and what are they like?

PRACTICAL ASSIGNMENT 9. SIMPLE SEARCH ALGORITHMS

Goal: to acquire skills and consolidate knowledge when performing search operations.

Preparatory work:

- data sets: arrays, linear lists;
- search algorithms by numerical key: linear, linear with barrier, binary, exponential, interpolation;
- Pattern search algorithms in text: brute force, Knuth-Morris-Pratt (KMP), Boyer-Moore (BM) algorithms.

General Information

Search involves finding a specified search key in a fixed set of data of the same type. Among the simple search algorithms, the following are known: linear search, linear search with a sentinel, binary search, exponential search, interpolation search.

When comparing different algorithms, it is important to know how their complexity depends on the size of the input data - this is the so-called time complexity of the algorithm. When determining the time complexity of search algorithms, the number of comparison and assignment operations is primarily considered.

Linear search algorithm. The simplest method of searching for an element in an unordered set of data, based on its key value, is to sequentially examine each element in the set until the desired element is found, for which $m[i]$ = key. If the entire set has been examined but the element is not found, it means that the desired key is absent in the set. On average, the sequential search requires $(N+1)/2$ comparisons, thus the algorithm has a linear order complexity - $O(N)$.

Example 1. Linear search function

Function code

```
int LinearSearch(int arr[], int key)
{   unsigned i=0;
    while(i<N && arr[i]!=key) i++;
    if (i==N) return -1; else return i;
}
```

Since the search terminates in case of a condition failure, the exit condition of the loop is as follows:

(i==n) or (m[I]==key).

If ***i = n***, then the key is not found. It is evident that the loop termination is guaranteed since i is incremented at each step. Therefore, after a finite number of steps, it will reach n even if no comparisons were made.

Linear search with a barrier algorithm. At each step of linear search, it is necessary to increment the index i and evaluate a complex logical expression. Can the search process be accelerated? The only solution is to simplify the logical expression by formulating a simple expression, while ensuring that there will always be a match.

Such a variant is implemented in the linear search with a barrier method. In this method, an additional element with a value of key is appended to the end of the array. It is called a barrier because it prevents accessing beyond the array boundaries. However, now the array is defined as int ***int m[0..n + 1],*** and the search function is shown in the following code example:

Example 2. Linear search function with a barrier

Function code

```
int LinearSearchWithBarrier(int arr[], int key)
{   unsigned i=0;
    arr[N]=key;
    while(arr[i]!=key) i++;
    if ( i == N) return  -1; else return i;
}
```

The exit condition of the loop is ***m[i] == key***; however, if the loop exits when ***i = n + 1,*** it means the key is not found.

Binary search algorithm. Another relatively simple method of accessing an element is the binary search method, which is performed on a pre-ordered sequence of elements. This type of search is called binary search, dichotomous search, or half-interval search. Entries in the table are arranged in ascending order, either lexicographically (for string keys) or numerically (for numeric keys). Any sorting method can be used to achieve ordering.

In this method, searching for a specific record with a defined key value is

similar to searching for a surname in a phone directory. Initially, a record in the middle of the table is approximately determined, and the value of its key is analyzed. If it is too large, the value of the key in the middle of the first half of the table is analyzed. This procedure is repeated in this half until the desired record is found. If the key value is too small, the key corresponding to the record in the middle of the second half of the table is tested, and the procedure is repeated in this half. This process continues until the desired key is found or an empty interval is encountered in which the search is performed.

In the binary search algorithm, two variables ***b, e*** are introduced, marking the left (younger) and right (older) sections of the array, where the necessary element can be found.

Example 3. Binary search function

Function code

```
bool BinSearch (int *arr, int key, int &m)
{  int b=0, e=N;                    // initial values of limits
   bool notFound=true;
   while(b <= e && notFound)        // until the interval narrows to 0
   {   m=(b+e)/2;           // the middle of the interval
       if (arr[m]< key) b=m+1; // search on the right side of the array
       else  if (arr [m] > key) e=m-1; // otherwise - in the left part
             else notFound=false;      // the search was successful
    return notFound;
}
```

In general, the choice of ***i*** can be arbitrary.

However, when searching for the middle value, half of the array is excluded from the search interval. To find the desired record in the table, in the worst case, it takes $\boldsymbol{log_2(N)}$ comparisons. Therefore, the order of the binary search algorithm is logarithmic - $\boldsymbol{O(log_2(N))}$**.** This is better than linear search.

The efficiency of the algorithm can be significantly improved by simplifying the loop condition. In this case, the search for the moment of matching the desired key in the set is abandoned. This requires more comparisons, but the efficiency gain at each step exceeds these losses. The fast algorithm is based on the fact that for the desired element of the array, the condition is satisfied that all keys to the left are smaller, and all keys to the right are larger or equal. The search ends when the first element with the specified key is found.

Example 4. Improved binary search algorithm

Function code

```
int BinSearchQuick (int *arr, int key)
{ int m, b=0, e=N;                  // initial values of limits
  while (b <= e)            // until the interval narrows to 0
  {  m = (b + e) / 2;               // the middle of the interval
     if (arr[m] >= key) e=m-1;
     else b=m+1;
  }
  if (arr[b]==key) return b;
  else return -1;
} }
```

It is obvious that the exit condition is reachable because for the middle value, the condition ***b<= i < e*** holds true. Therefore, ***i – 1*** decreases and ***i + 1*** increases. The loop ends when ***b = e***. However, this is not a guarantee of a successful search. An additional check is needed after the loop is completed:

m[i] == key.

Searching in an array is called table lookup when the key itself is an integrated object, such as an array of numbers or characters. The latter case is often encountered, where arrays of characters are referred to as strings or words. Table lookup requires "nested" searches, namely: searching within the rows of the table, and for each row, sequential comparisons are made between the components.

A typical task

Implement a linear key search algorithm in an array of integers.

Program code

```
#include <iostream>
#include <conio.h>
int linearSearch(int[],int,int);

int main(void)
{         setlocale(LC_CTYPE, "rus"); // для роботи з кирилицею
          const int arraySize=100;
          int a[arraySize],searchKey,element;
          for (int x=0;x<arraySize;x++)
                    a[x]=2*x;
          std::cout<<"Введіть ключ пошуку - ціле число:  ";
          std::cin>>searchKey;
          element=linearSearch(a,searchKey,arraySize);
          if (element!=-1)
                    std::cout<<"Знайдено значення в елементі "<<element<<std::endl;
          else
                    std::cout<<"Значення не знайдено"<<std::endl;
return 0;
}
 int linearSearch(int array[],int key,int sizeOfArray)
  {  for (int n=0; n<sizeOfArray;n++)
          if (array[n]==key)
             return n;
```

```
    return -1;
}
```

The result of the program:

```
Enter the search key - an integer:  17
No value found
```

```
Enter the search key - an integer:  98
A value is found in the element 49
```

Individual tasks

To develop and debug the program that implements two search algorithms according to the task. Compare the algorithms based on their runtime.

During the testing phase, for each of the algorithms (variants 4-15), determine the number of comparisons in the dataset with different numbers of elements (20, 100, 1000, 10000), measure the search time, fill in the table according to Table 9.1, create graphs, and draw conclusions. When working with text (variants 1-3), vary the length of the text and the pattern.

NOTE: Tasks 4-15 involve working with integers; the next 12 tasks (numbered 16-27) require working with real numbers.

1) Direct and KMP pattern searching in a text.
2) KMP and BM pattern searching in a text.
3) Direct and BM pattern searching in a text.
4) Binary and linear searches in an array.
5) Binary and linear searches in a linear list.
6) Linear with a barrier and binary searches in an array.
7) Linear with a barrier and binary searches in a linear list.
8) Linear and linear with a barrier searches in an array.

9) Linear and linear with a barrier searches in a linear list.

10) Exponential and interpolation searches in an array.

11) Linear search in an array and a linear list.

12) Linear with a barrier search in an array and a linear list.

13) Binary search in an array and a linear list.

14) Binary and interpolation searches in an array.

15) Binary and exponential searches in an array.

16) Linear with a barrier and exponential searches in an array.

17) Linear and exponential searches in an array.

Table 9.1 – Results of testing search algorithms

Number of elements	20	100	1000	10000
Number of comparisons				
Search time				

Checkpoint questions

1. What determines the complexity of the algorithm?

2. What condition must be satisfied when searching for an integer key?

3. What condition must be satisfied when searching for a floating-point key?

4. In the linear search with a barrier algorithm, what serves as the barrier?

5. List all known simple algorithms for searching by numeric key in descending order of their average search time.

6. What limitations are imposed on the data set in linear search with a barrier and without a barrier?

7. Explain how binary search is performed.

8. Which algorithm is implemented in the following code fragment?

```
{   while(m[i]!= key && i<N) i++;

    if( m[i]== key) return i; else return –1;

}
```

9. What limitations are imposed on the data in binary search?

10. Which algorithm is implemented in the following code fragment?

```
{   while(m[i]!= key) i++;

    if( i!= N) return i;  else return –1;

}
```

11. Draw a qualitative graph showing the dependency of the search time on the number of elements in the dataset for the following search algorithms: linear search, linear search with a barrier, binary search, interpolation search, and exponential search. Provide explanations.

12. Explain how exponential search is performed and how it differs from binary search.

13. Explain the algorithm of Boyer-Moore pattern searching in a text.

14. Explain the algorithm of Knuth-Morris-Pratt pattern searching in a text.

15. What is the significant difference between the direct pattern search algorithm and the KMP and BM search algorithms?

PRACTICAL ASSIGNMENT 10. SEARCH ALGORITHMS USING TABLES

Goal: to reinforce knowledge about search algorithms that require additional memory; acquire skills in performing search operations using direct access tables, dictionaries, and hashed tables.

Preparatory work:

- arrays and lists;
- files;
- simple search algorithms;
- search algorithms using direct access tables;
- concepts of hash tables, hashing functions, collisions;
- collision resolution algorithms.

General Information

Direct access table is the simplest table that provides a very fast search. The search key is the address of the record in the table, and the record is selected based on this address. If the selected record is empty, it means that there is no record with such a key in the table.

Hashed table is a table where the search key is transformed into the address of its record using a hash function. There is a possibility that different keys may be mapped to the same address of a record. This is known as a collision or overflow, and such keys are called synonyms. Collisions (also known as conflicts) are the main issue for hashed tables. A well-chosen hash function can minimize the number of collisions, but it cannot guarantee their complete absence. There are various methods to solve the collision problem in hashed tables, which are created based on arrays only, arrays and lists, or just lists.

Collision resolution algorithms. Rehashing, also known as open addressing, works as follows: if it is found that the required place in the table is already occupied during an attempt to insert into the table, the value is written to

any other place in the same table. The other place is determined using a secondary hash function *H*2, he argument of which can be the original key value or the result of previous hashing: $r = H2(k, r')$,

where r' is the address obtained from the previous application of the hash function.

If the address obtained from the application of the *H*2 function is also found to be occupied, the ***H2*** function is applied again until a free space is found. The simplest secondary hash function is: $r = r' + 1$.

This function is sometimes called linear probing. In practice, with linear probing, if the "legitimate" slot (i.e., the slot located at the address obtained from the primary hash function) is already occupied, the record occupies the first available place after the "legitimate" one (the table is considered as a circle).

Retrieving an element by key is done similarly:

– the address of the record is computed using the primary hash function and the record's key;

– the record located at the obtained address is read and compared with the search key;

– if the record is not empty and the keys do not match, the search continues using the secondary hash function.

The search ends when a matching record is found (successful completion) or when the entire table has been traversed (unsuccessful completion).

Clustering. The essence of the clustering method is that table records are combined into clusters of a fixed small size. The hash function outputs the address of the cluster, not the record. After finding the cluster, a linear search is performed within the cluster for the key. Clustering allows smoothing the non-uniform distribution of keys in the cluster space and, therefore, reduces the number of collisions, but it cannot guarantee to prevent them entirely. Clusters can also become full. Therefore, clustering is used as a complement to more radical methods, such as rehashing or other methods described below.

Hash tables with a common overflow area. In the case of a hash table with a common overflow area, two memory areas are allocated: the main area and the overflow area. The hash function outputs the address of the record (or cluster) in the main area. When inserting a record, if its "legitimate" place in the main area is already occupied, the record is placed in the first available space in

the overflow area.

During the search, if the "legitimate" place in the main area is occupied by a record with a different key, a linear search is performed in the overflow area. The common overflow area requires more memory than open addressing: while the size of an open-addressed table may not exceed the size of the actual record set, additional memory is needed here for the overflow area. However, the efficiency of accessing the table with an overflow area is higher than with double hashing. If a first attempt fails in a table with double hashing, and the search has to continue through the entire table, then in a table with an overflow area, the search continuation is limited only to the overflow area, the size of which is significantly smaller than that of the main table.

Hash tables with distributed overflow chains. It is natural to limit the search, in case of collisions, only to the set of key values that claim the same place in the main table. This idea is implemented in tables with overflow chains. Each record structure is extended with an additional field - a pointer to the next record. Through these pointers, records with synonym keys are linked in a linear list, the beginning of which is in the main table, and the continuation is outside of it. When inserting a record into the table, the address of the record (or cluster) in the main area is computed using the hash function. If the place in the main area is free, the record is inserted into the main table. If the place in the main area is occupied, the record is placed outside of it. Memory for such a record with a synonym key can be allocated dynamically for each new record, or a pre-allocated overflow area can be assigned for the synonym. After placing the synonym record, the pointer field from the main table record is transferred to the pointer field of the synonym, and in its place, a pointer to the just-placed synonym is written in the main table record.

Although tables with overflow chains increase the size of each record and slightly complicates processing due to pointer handling, narrowing the search area provides a significant gain in efficiency.

In any method of constructing hash tables, the problem of removing an element from the main area arises. The record to be removed must first be found in the table. If the record is found by secondary hashing (open addressing) or in the overflow area (table with a common overflow area), marking the record as empty is sufficient. However, if the record is found in a chain (table with overflow chains), the pointer of the previous element in the chain also needs to

be adjusted.

If the record to be removed is in its "legitimate" place, marking it as empty makes it impossible to search for its synonyms, which may still be in the table. One way to solve this problem is to mark the record with a special "deleted" flag. This method is often used in tables with double hashing and a common overflow area, but it does not free memory or speed up the search when reducing the number of elements in the table.

Another approach is to find any synonym of the record being removed and move it to the "legitimate" place. This approach is easily implemented in tables with chains but requires significant effort in tables with a different structure since a search through the entire open-addressed table or the entire overflow area is required, including the computation of the hash function for each element being checked.

A typical task

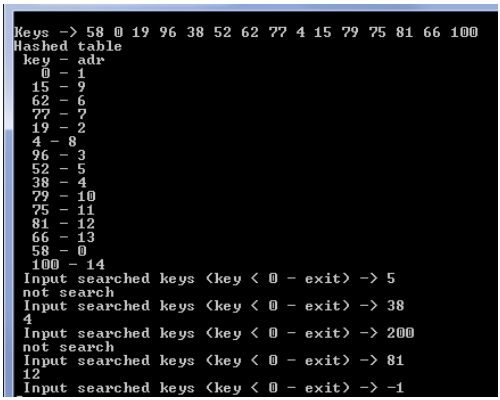
Task. Implement the algorithm for finding a key in an array of integers using a hash table. The hashing function is division by modulus. The collision resolution algorithm is open hashing with linear testing.

Program code

```
#include <iostream>
using namespace std;

void Init(void);
int Insert(int key, int adr);
int Search(int key);

#define N 15               // the number of entries in the table
#define EMPTY -1

struct ElHashTabl
{        int key;
         int adr;
};
ElHashTabl hashTabl[N];     // hash table
int keys[N]={58,0,19,96,38,52,62,77,4,15,79,75,81,66,100};

void main(void)
{    int i, key, res;
     Init();
     cout <<"\nKeys -> ";
     for (i=0;i<N;i++)
              cout << keys[i] <<" ";
     for (i=0;i<N;i++)
              Insert(keys[i], i);
```

```
     cout <<"\nHashed table \n key - adr \n ";
     for (i=0;i<N;i++)
             cout <<"  " <<hashTabl[i].key <<" - " <<hashTabl[i].adr << "\n" ;
          cout <<" Input searched keys (key < 0 - exit) -> ";
          cin >> key;
     while (key >=0)
     {   res=Search(key);
             if (res==EMPTY)          cout << " not search \n";
             else cout << " " <<res <<"\n" ;
        cout <<" Input searched keys (key < 0 - exit) -> ";
             cin >> key;
          }
}

int Hash(int key)           // hash function
{     return (key%N);
}
void Init(void)             // hash table initialization
{     for(int i=0;i<N;i++)
          hashTabl[i].adr=EMPTY;
}
int Insert(int key, int adr)         // adding the key to the hash table
{    int addr,a1;
     addr=Hash(key);
     if (hashTabl[addr].adr!=EMPTY)
     {   a1=addr;
         do
         {    addr=Hash(addr+1);
         }while(!((addr==a1)||(hashTabl[addr].adr==EMPTY)));
         if (hashTabl[addr].adr != EMPTY)
              return 0;
     }
     hashTabl[addr].key=key;
     hashTabl[addr].adr=adr;
     return 1;
}
int Search(int key)                  // key search function
{    int addr, a1;
     addr=Hash(key);
     if (hashTabl[addr].adr==EMPTY)
         return EMPTY;      // space is free - the key is not in the table
     else
         if (hashTabl[addr].key==key)
                  return hashTabl[addr].adr;  // the key is found in its place
              else                            // the place is occupied by another
key
              {   a1=Hash(addr+1);
          // Search until a key is found or a full rotation is made
                  while((hashTabl[a1].key != key)&&(a1!=addr))
                       a1=Hash(a1+1);
                       if (hashTabl[a1].key != key)
                           return EMPTY;
                       else
                           return hashTabl[a1].adr;
              }
}
```

A fragment of the results of the program



Individual tasks

The initial data is contained in a text file. Read the file, create a direct access table or a hash table according to the task in Table 10.1. Verify the functionality of the created tables through search operations.

Compare the search time using the created tables with simple search algorithms from practical assignment 4. For each of the algorithms, determine the number of comparisons in a dataset with different numbers of elements (20, 1000, 5000, 10000, 50000), calculate the search time, fill in the table following the format in Table 10.2, create graphs, and draw conclusions.

Table 10.1 – Initial data for the practical assignment

№	The content of the initial data	Table	Hash function	Search key
1	2	3	4	5
1	Record book number, student's name, the average score	Hash table with double hashing	Division by modulus; rehashing: Address = Address + 1	The average score

2	Reader's card number, reader's last name, number of books	Clustering hash table.	Folding function and modulo division	Reader's last name
3	Processor model, motherboard type, hard drive volume, RAM size	Hash table with a common overflow area.	Division by modulus; rehashing: Address = Address + 1	Hard drive volume/ RAM size
4	Code, engine type, power, number of revolutions	Direct access table		Code
5	The station's network address, operating system, network protocol	Hash table with distributed overflow chaining.	Number system conversion function and modulo division	The station's network address
6	Code, type of aircraft, speed, number of passengers	Hash table with a common overflow area.	Mid-square function	Code
7	Account card number, person's last name, age, home address.	A hash table with distributed overflow chains	The conversion function of the counting system and division by modulo	Account card number
8	Product code, name, price, quantity	Direct access table		Product code
9	Identification code, network user name, E-mail address	A hash table with distributed overflow chains	Mid-square function and modulo division	Identification code
10	City code, city name, area, population.	Hash table with double hashing	Folding function and modulo division	City code

11	Country code, country name, capital city.	Hash table with double hashing	Mid-square function	Country code
12	Employee ID, last name of the employee, department or division.	A hash table with distributed overflow chains	Folding function and modulo division	Employee ID
13	Phone number, owner`s name, address, time of conversation	A hash table with distributed overflow chains	Mid-square function	Phone number
14	File name, size, date of creation, date of last modification	Hash table with clustering and double hashing.	Folding function and modulo division, rehashing	File name
15	Book code, author's name, number of copies	Hash table with a common overflow area.	Folding function and modulo division	Book code
16	Name of the university, city, number of faculties, number of students	A hash table with distributed overflow chains	Folding function and modulo division	Name of the university

Table 10.2 – Results of testing search algorithms using tables

Number of elements	20	1000	5000	10000	50000
Number of comparisons					
Sorting time					

Checkpoint questions

1. What is written in the hash table?

2. What determines the index of a record in a hash table?

3. hat are the main problems in hashing and what do they involve?

4. How many comparison operations are performed when searching for a key using direct access tables?

5. What keys are called synonyms?

6. What are the disadvantages of the key search algorithm using direct access tables?

7. What is the purpose of a hash function?

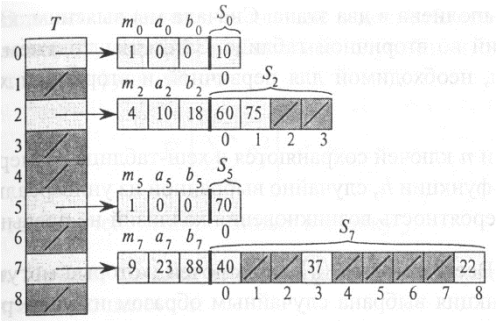
8. Name the basic hashing functions.

9. How is a hash table created when implementing open addressing? What is stored in the elements of such a table?

10. How is a direct access table created? What is stored in its elements?

11. Name the algorithms for resolving collisions in open addressing.

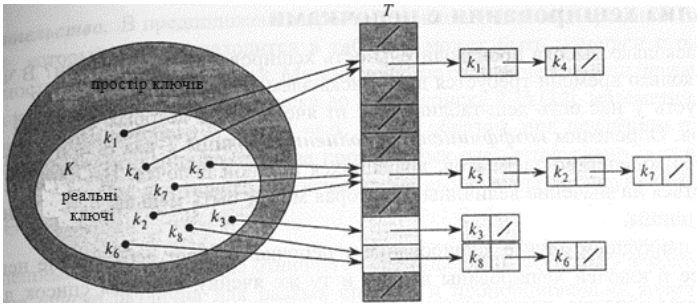
12. What type of hashing is shown in this figure?



13. What is the essence of the method of resolving collisions using

distributed overflow chains?

14. Which hashing method is shown in the figure?



15. Why is searching using direct access tables not widely adopted?

PRACTICAL ASSIGNMENT 11. SELECTION AND INCLUSION SORTING ALGORITHMS

Goal: to reinforce theoretical knowledge and gain practical experience in sorting a set of static and dynamic data structures.

Preparatory work:

- arrays and lists;
- classification of sorting algorithms;
- selection sort and insertion sort algorithms.

General Information

The purpose of sorting is to transform an input sequence into a sequence that contains the same records but in ascending (or descending) order of the key values. A sorting method is considered stable if it maintains the relative order of records with equal key values.

Sorting algorithms are characterized by their complexity order (quadratic - $O(N^a)$, linear – $O(N)$, logarithmic – $O(N\log(N))$, memory requirements, sensitivity to the initial order of the input data, and complexity.

Sorting algorithms can be classified into four groups based on their working logic: selection, insertion, distribution, and merge.

Selection sort. The general algorithmic description is as follows: from the input set, the next element according to the order criterion is selected and included in the output set at the next position.

The following algorithms belong to this group: simple selection sort, exchange selection sort, with search for maximum and minimum elements, classical bubble sort, modified bubble sort, with a flag, with memorizing the location of the last exchange, Cocktail Shaker sort, Shell sort. All of them are characterized by order $O(N^a)$.

Insertion sort. The general algorithmic description is: from the input set, the next element according to its position number is selected and included in the output set at the (previously vacated) position it should occupy according to the

key order.

The following algorithms belong to this group: simple insertion sort, exchange insertion sort, bubble sort with insertions, insertion sort with a barrier, binary insertion method. These algorithms are characterized by order $O(N^a)$.

More complex algorithms such as tournament sort, heap sort, and sorting with partially ordered trees also belong to the insertion sort group. These algorithms are characterized by order O(Nlog(N)). They can partially be attributed to the distribution sort group, and therefore, they will be considered in the next practical assignment.

A typical task

Task: Develop a program to sort an array of integers according to the Shaker and insertion sort algorithms with a barrier.

Program code

```
#include<iostream>
#include<conio.h>
#define N 20                // number of elements in the array

void Create(int []);
void CopyArray(const int [], int []);
void OutConsole(int []);
void SortShaker(int []);
void BarierInsertionSort(int []);

int main()
{
    int array[N], copyArray[N];
    Create(array);
    std::cout << "Initial array \n";
    OutConsole(array);
    CopyArray(array, copyArray);
    SortShaker(array);
    std::cout << "Sorted array \n";
    OutConsole(array);
    CopyArray(copyArray, array);
    std::cout << "Initial array \n";
    OutConsole(array);
    BarierInsertionSort(array);
    std::cout << "Sorted array \n";
    OutConsole(array);
}

void Create(int a[])
{
    for(int i = 0; i < N; i++)
        a[i] = rand() % 100;
```

```
}

void CopyArray(const int a[], int copy_a[])
{
    for(int i = 0; i < N; i++)
        copy_a[i] = a[i];
}

void OutConsole(int a[])
{
    for(int i = 0; i < N; i++)
        std::cout << "  " << a[i];
    std::cout << "\n";
}

void SortShaker(int a[])
{
    int temp;
    int key = 0; // key - set to 1 if no swaps were made (array is sorted)
    int i, j;
    for (i = 1; i < N-1; i++)
    {
        if (key) break;
        key = 1;
        for (j = i; j < N-i; j++)
        {
            if (a[j-1] > a[j])
            {
                key = 0;
                temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
        for (; i; i--)
        {
            if (a[j] < a[j-1])
            {
                key = 0;
                temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}

void BarierInsertionSort(int a[])
{
    int i, j, k, t;
    for (i = 1; i < N; i++)  // Iterating over the input array - [i..N], output set
- [1..i]
    {
        a[0] = a[i];   // Setting up a barrier a[0]
        t = a[i];      // Remembering the value of the new element
        j = i - 1;
        while (a[j] > t) // Finding the position for the element in the output set
with shifting
        {
```

```
            a[j+1] = a[j];  // Shifting all elements greater than the new element
            j--;      // Looping from the end to the beginning of the output set
        }
        a[j+1] = t;       // Placing the new element in its correct position
    }
}
```

The result of the program:

```
Initial array
  83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36
Sorted array
  15  21  26  26  27  35  36  40  49  59  62  63  72  77  83  86  86  90  92  93
Initial array
  83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36
Sorted array
  36  15  21  26  26  27  35  36  40  49  59  62  63  72  77  86  86  90  92  93
```

Individual tasks

Write a program that implements the sorting of a static and/or dynamic set of data in a given way according to the data in the table. 11.1.

Determine the number of comparisons and exchanges for data sets containing different numbers of elements (20, 1000, 5000, 10000, 50000).

Estimate the sorting time. Investigate the effect of the initial ordering of the data set (sorted, reverse-sorted, random).

Record all the obtained data in table 11.2. Draw conclusions.

Table 11.1 – Variants of individual tasks

N	Algorithm number	N	Algorithm number	N	Algorithm number	N	Algorithm number
1	13	7	10	13	7	19	4
2	1, 10	8	5, 9	14	8,1	20	11, 13
3	12	9	9	15	6	21	3
4	3, 7	10	6, 2	16	9, 13	22	10, 6
5	11	11	8	17	5	23	2
6	4, 8	12	7, 4	18	2, 3	24	4, 10

where N is the individual task number.

Note:

1) For odd N, implement the specified sorting algorithm on static and dynamic data sets.

2) For even N, implement two sorting algorithms on one of the data sets (static or dynamic).

3) Choose the sorting algorithm by number from the following list of sorting algorithms.

Sorting algorithms:

1. Simple selection sort.
2. Exchange selection sort.
3. Selection sort with finding maximum and minimum elements.
4. Bubble sort with a flag.
5. Modified bubble sort.
6. Bubble sort with memorizing the location of the last exchange.
7. Shaker sort.
8. Shell sort.
9. Gnome sort.
10. Simple insertion sort.
11. Bubble sort with insertions.
12. Insertion sort with a barrier.
13. Binary insertion sort.

Table 11.2 – Comparative efficiency table form for the algorithm.

Sorted data set	20	1000	5000	10000	50000

Number of comparisons					
Number of transfers					
Sorting time					

Note: fill in the table also for a *random set of data and sorted in reverse order*

Checkpoint questions

1. Name all sorting algorithms of orders $O(n^2)$, O(n), O(log(n)).
2. Name all sorting algorithms belonging to the bubble sort group. What characterizes
3. What sorting algorithms are called stable?
4. What operations (except for comparison operations) have a great impact on the execution time of sorting?
5. What factors affect the choice of a sorting algorithm?

PRACTICAL ASSIGNMENT 12. ALGORITHMS OF SORTING BY DISTRIBUTION AND MERGING

Goal: consolidate theoretical knowledge and gain practical experience in organizing a set of static and dynamic data structures.

Preparatory work:

- arrays and lists;

- selection sort and insertion sort algorithms;

- tree-based selection sort, distribution sort, and merge sort algorithms.

General Information

In the previously considered sorting algorithms, the search for the minimum element from N elements, then from (N – 1), then from (N – 2) and so on to the last element was performed by selection. As a result, the number of comparisons is (N2 – N)/2.

Selection Sort. In the selection sort algorithms considered in this work, the general algorithm is as follows: among N elements of the array, the smaller element is determined in each pair of adjacent elements. This process requires N/2 comparisons, and there will be N/2 minimum elements. Then, from these elements, further minimum elements are determined, but now there are N/4, and so on, until the smallest element is found. This is the first stage - building the tree.

The second stage involves traversing the tree, selecting an element with the smallest key, and reorganizing the tree. This is how the tournament sort algorithm works.

Heap Sort is based on building a heap. Floyd proposed a sorting algorithm consisting of two stages. In the first stage, a heap is constructed based on the array, and the largest element is moved to the top of the heap. In the second stage, the largest element is recorded at the end of the input portion, and the heap is reorganized. The time complexity of this algorithm is O(Nlog(N)).

Distribution Sort. The general algorithm works as follows: the input set is

divided into several subsets (possibly of smaller size), and sorting is performed within each subset. This group of algorithms includes radix sort, quicksort (Hoare Partition Scheme), bucket sort, degenerate distribution sort.

Merge Sort. The general algorithm works as follows: the input set of N elements is treated as N ordered sets, each containing one element. These sets are merged into ordered sets of two elements, then four elements, and so on. The output set is formed by merging smaller ordered subsets. The algorithms in this group include straight merge sort and pairwise merge sort. These algorithms have a time complexity of O(Nlog(N)).

A typical task

Task: Implement the algorithm for Heap Sort. Verify its functionality.

Program code

```
#include<iostream>
#include<conio.h>
#define N 40               // number of elements in the array

void Create(int []);
void OutConsole(int []);
void HeapSort(int []);
#define swap(x,y)  {int t=x; x=y; y=t;}

void main()
{
    setlocale(LC_CTYPE, "rus");      // for working with Cyrillic characters
    int array[N];
    Create(array);
    std::cout << "Initial array \n";
    OutConsole(array);
    HeapSort(array);
    std::cout << "Sorted array \n";
    OutConsole(array);
}

void Create(int a[])
{
    for(int i = 0; i < N; i++)
       a[i] = rand() % 100;
}

void OutConsole(int a[])
{
    for(int i = 0; i < N; i++)
             std::cout << "  " << a[i];
    std::cout << "\n";
}
```

```
void downHeap(int a[], int k, int n) // Procedure for sifting the next element
{ // Before the procedure: a[k+1]...a[n] - a heap
  // After: a[k]...a[n] - a heap
  int new_elem;
  int child;
  new_elem = a[k];
  while(k <= n/2)                    // while a[k] has children
  {
    child = 2*k;
    // choose the larger child
    if( child < n && a[child] < a[child+1] )
      child++;
    if( new_elem >= a[child] ) break;
    a[k] = a[child];        // otherwise, move the child up
    k = child;
  }
  a[k] = new_elem;
}

// Heap sort
void HeapSort(int a[])
{
  int i;
  // Build the heap
  for(i = N/2-1; i >= 0; i--)
    downHeap(a, i, N-1);
  // Now a[0]...a[size-1] is a heap
  for(i = N-1; i > 0; i--)
  {
    // swap the first with the last
    swap(a[i], a[0]);
    // restore the heap property a[0]...a[i-1]
    downHeap(a, 0, i-1);
  }
}
```

The result of the program:

```
Initial array
  83  86  77  15  93  35  86  92  49  21  62  27  90  59  63  26  40  26  72  36  11  68  67  29  82  30  62  23  67
  35  29  2  22  58  69  67  93  56  11  42
Sorted array
  2  11  11  15  21  22  23  26  26  27  29  29  30  35  35  36  40  42  49  56  58  59  62  62  63  67  67  67  68
 69  72  77  82  83  86  86  90  92  93  93
```

Individual tasks

Write a program that implements three sorting algorithms for a dataset according to Table 12.1.

Determine the number of comparisons and exchanges for initial datasets containing different numbers of elements (50, 1000, 5000, 10000, 50000).

Evaluate the sorting time. Investigate the influence of the initial order of

the dataset (sorted, reverse sorted, random).

Record all the obtained data in Table 12.2. Draw conclusions.

Table 12.1 – Variants of individual tasks

N	Algorithm number	N	Algorithm number	N	Algorithm number	N	Algorithm number
1	1, 4, 9	7	2, 6, 9	13	4, 1, 8	19	5, 1, 8
2	1, 4, 5	8	2, 7, 8	14	4, 2, 8	20	5, 2, 9
3	1, 6, 8	9	3, 4, 8	15	4, 3, 8	21	5, 3, 9
4	1, 7, 9	10	3, 5, 9	16	4, 1, 9	22	5, 4, 8
5	2, 4, 6	11	3, 7, 9	17	4, 2, 9	23	5, 7, 9
6	2, 5, 7	12	3, 6, 9	18	4, 3, 5	24	5, 6, 9

Note:

1) N – the individual task number.

2) Choose the sorting algorithm from the following list.

Sorting Algorithms:

1. Tournament Sort.
2. Heap sort.
3. Partially Ordered Tree Sort
4. Radix Sort
5. Quicksort
6. Bucket Sort
7. Degenerate Distribution Sort
8. Merge Sort
9. Pairwise Merge Sort

Table 12.2 – The form of the comparison table of the efficiency of the algorithm

Sorted data set	20	1000	5000	10000	50000
Number of comparisons					
Number of transfers					
Sorting time					
Sorted in reverse order	20	1000	5000	10000	50000
Number of comparisons					
Number of transfers					
Sorting time					
A random data set	20	1000	5000	10000	50000
Number of comparisons					
Number of transfers					
Sorting time					

Checkpoint questions

1. Explain why tree-based sorting algorithms have a time complexity of $Nlog_2(N)$?

2. What is the main essence of the Quicksort algorithm?

3. Which sorting algorithms require additional memory and how is it used?

4. Name the sorting algorithms that do not involve comparisons between elements in the dataset being sorted.

5. What factors influence the choice of a sorting algorithm?

6. Provide the code for "Bucket Sort" algorithm with a single memory partition.

7. In how many steps will the following set of numbers be sorted in ascending order: 231, 24, 3, 35, 15, 932, 176, 83, 8, 54, 112?

8. What is the purpose of the additional memory required by the Quicksort algorithm?

9. What is the purpose of the additional memory required in substring search algorithms? Name these algorithms.

10. Provide a description of the Tournament Sort algorithm.

11. Provide a description of the Partially Ordered Tree Sort algorithm.

12. Provide a description of the Merge Sort algorithm.

13. Provide a description of the Pairwise Merge Sort algorithm.

14. What method is used to build a heap structure in the form of an array without explicitly constructing a tree?

15. Provide a description of the Heapsort algorithm.

16. Which sorting algorithms are considered stable?

PRACTICAL ASSIGNMENT 13. TREE PROCESSING ALGORITHMS

Goal: Gain practical experience working with binary trees.

Preparatory work:

- Non-linear lists.
- Graphs.
- Trees.
- Operations on trees.

General Information

A tree is a graph in which there is a unique vertex that is not referenced by any other vertex (root). Each vertex, except the root, has only a single reference, and there are vertices that are not referenced by any other vertices (leaves). In trees, vertices are commonly referred to as nodes. The number of edges leaving a vertex in a graph is called the outdegree of that vertex. Trees in which the outdegree of each vertex $\leq$ m (where m can be 0, 1, 2, 3, etc.) are called m-ary trees. Trees with m = 2 are called binary trees.

A forest consists of multiple trees that form a single system and are considered together. Forests and trees of any arity can be transformed into binary trees according to certain rules.

Trees can be represented using arrays (sequential representation) or non-linear lists (linked representation). In the linked representation of binary trees, a structure is created to represent each node, which has an information field (which can be a structure itself) and two pointer fields for the left and right subtrees (Figure 13.1).

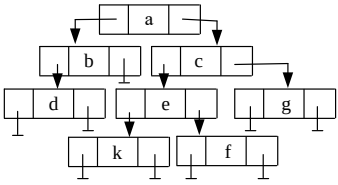


Figure 13.1 - Linked representation of a tree using a non-linear list.

Binary search trees are trees in which the relationship between key values in any branching node and its descendants, as well as among the descendants themselves, is maintained.

Types of trees: perfectly balanced, balanced, and unbalanced.

Operations on trees: tree traversal (preorder, mixed, postorder), insertion, deletion, node search, tree balancing.

Tree processing functions are typically recursive. To enable the implementation of non-recursive tree functions, trees are threaded.

Tree threading involves replacing empty pointers in tree nodes with specific values (pointing to the next nodes) based on the traversal rule. Two fields are added to the node structure to describe the connections. Threading complicates the implementation of node insertion and deletion operations.

A typical task

Develop a program that allows creating a binary tree with non-repeating keys. Output the contents of the tree on the screen. The values of the data in the tree nodes can be entered from the keyboard or generated according to the user's decision.

Program code

```
#include <conio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;

class CtreeNode {
         int key;                              // key
         int info;                   // information field
         CtreeNode* LLink, * RLink; // pointer to left and right node
public:
         CtreeNode() {
                  key = 0;
                  LLink = RLink = NULL;
         } // default constructor

         CtreeNode(int k, int info_c) {
                  key = k;
                  info = info_c;
                  LLink = RLink = NULL;
         } // constructor with parameters

         ~CtreeNode() {} // Destructor
```

```
        // function to add an element to the tree. Returns a pointer to the root
        void Insert(int number, int info, CtreeNode* ptr);
        void PrintTree(CtreeNode* ptr, int n);
        void DeleteTree(CtreeNode* ptr);
};

void CtreeNode::Insert(int n, int inf, CtreeNode* ptr) {
        cout << "\n Inserting element (" << n << ";" << inf << ")\n";
        int flag = 1;
        CtreeNode* p, * q;
        p = ptr;
        while (flag) {
                if (n < p->key) {
                        q = p->LLink;
                        cout << "Found node " << p->key << endl << "Going left
";
                        if (q == NULL)  cout << "NULL" << endl;
                        else cout << q->key << endl;
                        if (q == NULL) {
                                flag = 0;
                                cout << "Creating new element (" << n << "," <<
inf << ")\n";
                                q = new CtreeNode(n, inf);
                                cout << "Updating links: " << p->key << "-
>LLink = " << q->key << endl;
                                p->LLink = q;
                        }
                        else p = q;
                }
                else if (n > p->key) {
                        q = p->RLink;
                        cout << "Found node " << p->key << endl << "Going right
";
                        if (q == NULL) cout << "NULL" << endl;
                        else cout << q->key << endl;
                        if (q == NULL) {
                                flag = 0;
                                cout << "Creating new element (" << n << "," <<
inf << ")\n ";
                                q = new CtreeNode(n, inf);
                                cout << "Updating links: " << p->key << "-
>RLink = " << q->key << endl;
                                p->RLink = q;
                        }
                        else p = q;
                }
                else if (n == p->key) {
                        cout << "This element already exists" << endl;
                        flag = 0; // exit the WHILE loop
                }
        }
}

void CtreeNode::PrintTree(CtreeNode* ptr, int n) {
        if (ptr) {
                PrintTree(ptr->RLink, n + 3);
                for (int i = 1; i < n; i++)
                        cout << " ";
```

```
                cout << ptr->key << endl;
                PrintTree(ptr->LLink, n + 3);
        }
}

void CtreeNode::DeleteTree(CtreeNode* ptr) {
        if (ptr != NULL) {
                DeleteTree(ptr->LLink);
                DeleteTree(ptr->RLink);
                delete(ptr);
        }
}

int main() {
        int k_elem;
        char rand_elem;
        cout << "How many elements to add? ";
        cin >> k_elem;
        cout << "Generate random elements? (y/n) ";
        cin >> rand_elem;
        int num, inf;
        if (rand_elem == 'n') {
                cout << endl << "Index: ";  // input from the keyboard
                cin >> num;
                cout << "Value: ";
                cin >> inf;
        }
        else {
                num = rand() % 100 + 1; // random values
                inf = rand() % 100 + 1;
        }
        CtreeNode* root = new CtreeNode(num, inf); // pointer to the root
        for (int tr_i = 1; tr_i < k_elem; tr_i++) {
                if (rand_elem == 'n') {
                        cout << endl << "Index: ";  // input from the keyboard
                        cin >> num;
                        cout << "Value: ";
                        cin >> inf;
                }
                else {
                        num = rand() % 100 + 1; // random values
                        inf = rand() % 100 + 1;
                }
                root->Insert(num, inf, root);
        }
        root->PrintTree(root, 5);
        root->DeleteTree(root);             // freeing memory
        return 0;
}
```

The result of the program:

```
How many elements to add? 5
Generate random elements? (y/n) y

 Inserting element (78;16)
Found node 84
Going left NULL
Creating new element (78,16)
Updating links: 84->LLink = 78

 Inserting element (94;36)
Found node 84
Going right NULL
Creating new element (94,36)
 Updating links: 84->RLink = 94

 Inserting element (87;93)
Found node 84
Going right 94
Found node 94
Going left NULL
Creating new element (87,93)
Updating links: 94->LLink = 87

 Inserting element (50;22)
Found node 84
Going left 78
Found node 78
Going left NULL
Creating new element (50,22)
Updating links: 78->LLink = 50
       94
          87
    84
       78
          50
```

Individual tasks

Develop a program that allows creating a binary tree and solving an individual task. For tasks 7-16 (where traversals are not specified), implement two traversal algorithms of your choice.

Store integer keys in the information part of the tree nodes. Output the contents of the tree and the results of the individual task on the screen.

1. Find the node with the ***max*** key; use postorder and mixed traversals of the tree using a stack.

2. Find the node with the ***min*** key; use postorder and mixed traversals of the tree using a stack.

3. Determine the number of nodes with the ***max*** key; use a recursive algorithm with mixed and preorder traversals.

4. Determine the number of even nodes; use mixed and postorder traversals using a stack.

5. Develop a program for tournament sorting.

6. Determine ***K***, the number of nodes whose key is greater than the given number ***N***; use a recursive algorithm with postorder traversal of the tree using a stack. Add a node with the key ***K***.

7. Develop a program to calculate the height of the tree.

8. Delete nodes with keys equal to the given number ***N***.

9. Create two trees: copy only even keys to one tree and odd keys to the other.

10. Insert a node with key ***N*** after the first node with key ***K***. Enter the values of ***K*** and ***N*** from the keyboard..

11. Delete nodes with odd keys.

12. Determine the arithmetic mean of even and odd keys in the tree.

13. Determine the number of leaf nodes in the tree.

14. Determine the number of internal nodes in the tree.

15. Determine the number of nodes that have only a left link.

16. Determine the number of nodes that have only a right link.

17. Compare the time of preorder and postorder tree traversals.

18. Compare the time of preorder and mixed tree traversals.

19. Compare the time of mixed and postorder tree traversals.

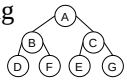
20. Thread the tree for its postorder traversal. Implement the postorder traversal of the tree. Compare the time of traversal for threaded and non-threaded trees.

21. Thread the tree for its mixed traversal. Implement the mixed traversal of the tree. Compare the time of traversal for threaded and non-threaded trees.

22. Thread the tree for its preorder traversal. Implement the preorder traversal of the tree. Compare the time of traversal for threaded and non-threaded trees.

Checkpoint questions

1. What is a tree?
2. How is the arity of a tree determined?
3. What operations are typical for trees?
4. What is tree traversal, and what are the known traversal operations?
5. What is a binary search tree?
6. Write the structure of the m-ary tree according to your discretion. Convert it to binary.
7. What are the different types of trees?
8. Describe the algorithm for preorder traversal of a tree.
9. Describe the algorithm for postorder traversal of a tree.
10. Describe the algorithm for inorder traversal of a tree.
11. Provide the sequence of nodes for the tree during postorder, preorder, and inorder traversals.



12. What is a full m-ary tree?
13. What is a perfectly balanced tree?
14. What is a full binary tree?
15. Given the arithmetic expression $((a + b) \cdot (c – d) + f) \cdot k$. construct the corresponding binary tree.
16. Write the arithmetic expression at your discretion. Create the corresponding tree for it. Perform a tree traversal and write the prefix, infix, and postfix notations for the given expression.
17. Which tree traversal function is provided in the code example?

```
void Order (Tree *t)
{  if (!t) return;
    if (t->left) Order(t->left);
     . . .  // Tree node data processing
    if (t->right) Order(t->right);   }
```

18. What are red-black trees, and what are their properties?
19. What is the purpose of tree balancing operations?
20. What properties characterize AVL trees?
21. What properties do perfectly balanced trees have?

PRACTICAL ASSIGNMENT 14. USE OF TREES

Goal: to gain practical experience in solving problems using binary trees.

Preparatory work:

- balancing trees;
- AVL trees;
- red-black trees;
- Huffman trees;
- trees for arithmetic expressions.

General Information

One of the methods that improve the search time in a binary tree is to create balanced trees that have minimal search time. A search tree is perfectly balanced if the number of nodes in the left and right subtrees of each node differs by no more than 1. During the correction of the tree, its ideal balancing is often violated. A more "soft" definition of balance is introduced for AVL trees: a search tree is balanced if the height of the left and right subtrees of any node differs by no more than 1. In such trees, the need for balancing operations occurs less frequently, and the balancing operations themselves are quite simple.

The balance in AVL trees is achieved through transformations called rotations. There are four types of rotations: L-rotation, R-rotation, LR-rotation, and RL-rotation. The first two are considered single rotations, and the last two are double rotations. This is related to the number of operations required to restore balance. Inserting a new key into an AVL tree or deleting a node can improve the balance or lead to its violation. In such cases, balancing of the current node is performed.

Nodes in an AVL tree are described by the following structure:

```
struct node

{  int key;

   unsigned char height;

   node* left;

   node* right;

};
```

The field *key* stores the node's key, the field *height* stores the height of the subtree with the root in this node, the fields *left* and *right* are pointers to the left and right subtrees. The constructor creates a new node (of height 1) with the given key *k*.

The red-black tree is one of the self-balancing binary search trees that guarantees logarithmic height growth based on the number of nodes, where basic tree operations such as insertion, deletion, and search for a node are performed quickly. Balance is achieved by introducing an additional attribute into the node structure called "color." This attribute can have one of two possible values: "black" or "red."

```
struct Node
{
        Node* left; // left descendant
        Node* right; // right descendant
        Node* parent; // parent node
        nodecolor color; // node color (BLACK, RED)
        int key; // key
};
```

Trees have a wide range of applications in translator implementations, working with arithmetic expressions, creating and maintaining symbol tables, databases, communication systems, etc.

Minimum coding trees are used for creating a code table. Usually, codes for symbols are created based on the frequency of their occurrence in the entire set of messages. Combining two symbols into one is performed using a binary tree structure. Each node contains a symbol and its frequency of occurrence. The code for any symbol in the alphabet can be determined by traversing the tree from the bottom up, starting from the leaf. Whenever a node is passed through, a 0 is appended to the code if going left and a 1 if going right. Once the tree is constructed, the code for any symbol in the alphabet can be determined

by traversing the tree from the bottom up, starting from the position that represents this symbol.

With the help of trees, any arithmetic expression can be represented. Each leaf in such a tree corresponds to an operand, and each parent node corresponds to an operation. Traversing the tree allows building any form of arithmetic expression notation: prefix, infix, or postfix.

A typical task

Develop a program that provides the creation of an AVL search tree, deletion of nodes by key value, and its balancing. Output the contents of the tree to the screen. Data values in nodes are defined in the program.

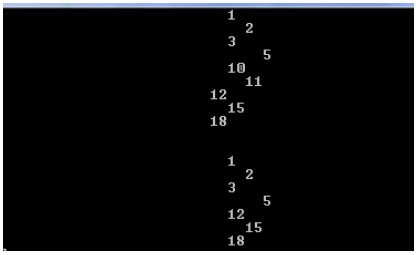
Program code

```
#include "stdlib.h"
#include "stdio.h"

struct node // structure for representing tree nodes
{
	int key;
	unsigned char height;
	node* left;
	node* right;
};
unsigned char Height(node* p)
{
	return p? p->height : 0;
}
int Bfactor(node* p)
{
	return Height(p->right) - Height(p->left);
}
void FixHeight(node* p)
{
	unsigned char hl = Height(p->left);
	unsigned char hr = Height(p->right);
	p->height = (hl > hr? hl : hr)+1;
}
node* RotateRight(node* p) // right turn around p
{	node* q = p->left;
	p->left = q->right;
	q->right = p;
	FixHeight(p);
	FixHeight(q);
	return q;
}
node* RotateLeft(node* q) // left turn around q
{	node* p = q->right;
	q->right = p->left;
	p->left = q;
	FixHeight(q);
```

```
        FixHeight(p);
        return p;
}
node* Balance(node* p) // balancing node p
{       FixHeight(p);
        if( Bfactor(p)==2 )
        {       if( Bfactor(p->right) < 0 )
                        p->right = RotateRight(p->right);
                return RotateLeft(p);
        }
        if( Bfactor(p)==-2 )
        {       if( Bfactor(p->left) > 0  )
                        p->left = RotateLeft(p->left);
                return RotateRight(p);
        }
        return p; // no balancing required
}
node* Insert(node* p, int k) // inserting key k into tree with root p
{       if( !p )
        {       p = new node;
                p->key = k; p->left = p->right = 0; p->height = 1;
                return p;
        }
        if( k<p->key )
                p->left = Insert(p->left,k);
        else
                p->right = Insert(p->right,k);
        return Balance(p);
}
node* FindMin(node* p) // search for the node with the minimum key in the tree p
{       return p->left? FindMin(p->left):p;
}
node* RemoveMin(node* p) // removing the node with the minimum key from the tree p
{       if( p->left==0 )
                return p->right;
        p->left = RemoveMin(p->left);
        return Balance(p);
}
node* Remove(node* p, int k) // deleting key k from tree p
{       if( !p ) return 0;
        if( k < p->key )
                p->left = Remove(p->left,k);
        else if( k > p->key )
                p->right = Remove(p->right,k);
        else //  k == p->key
        {       node* q = p->left;
                node* r = p->right;
                delete p;
                if( !r ) return q;
                node* min = FindMin(r);
                min->right = RemoveMin(r);
                min->left = q;
                return Balance(min);
        }
        return Balance(p);
}
void Print(node *tree, int level)
{       int i;
        if (tree == NULL)  return;
```

```
        Print(tree->left, level - 2);
        for (i = 0; i < level; i++)
                printf(" ");
        printf("%d\n", tree->key);
        Print(tree->right, level - 2);
}
void Delete(node *tree)  // removing the tree
{       if (tree == NULL) return;
        Delete(tree->left);
        Delete(tree->right);
        free(tree);
}

int main()
{       node *tree = NULL;
        tree = Insert(tree, 10);    // building a tree
        tree = Insert(tree, 5);
        tree = Insert(tree, 3);
        tree = Insert(tree, 1);
        tree = Insert(tree, 2);
        tree = Insert(tree, 11);
        tree = Insert(tree, 12);
        tree = Insert(tree, 15);
        tree = Insert(tree, 18);
        Print(tree, 30);
        printf("\n\n");
        Remove(tree, 10);
        Remove(tree, 11);
        Print(tree, 30);
        Delete(tree);
        return 0;
}
```

The result of the program:



Individual tasks

Develop a program that creates a binary tree and solves an individual task. Display the contents of the tree and the results of the individual task on the

screen.

1. Calculate the value of an arithmetic expression. Store the operands and arithmetic operations of the expression in the nodes of the tree in reverse Polish notation.

2. Build an AVL tree. Determine the value of the maximum key in the tree and remove all nodes whose key values are equal to the maximum.

3. Build a red-black tree. Enable the removal of any number of nodes based on a given value.

4. Build a Huffman tree for encoding a single message. Display the obtained codes for the characters and the encoded message.

5. Build a red-black tree. Determine the height of the tree. Add a node with a key value equal to the height of the tree.

6. Build an AVL tree. Enable the removal of any number of nodes based on given values.

7. Build an AVL tree. Enable the addition of any number of nodes to the tree.

8. Build a red-black tree where identical keys can be repeated. Determine how many times each key has been repeated in the tree.

9. Develop a program to build a Huffman tree for encoding your surname and first name. Display the obtained codes for the characters.

10. Build an AVL tree. Determine the value of the minimum key in the tree; remove all nodes whose key values are equal to the minimum.

11. Build an AVL tree where identical keys can be repeated. Determine how many times each key has been repeated in the tree.

12. Build an AVL tree. Determine the height of the tree. Add a node with a key value equal to the height of the tree.

Checkpoint questions

1. What are red-black trees? What are their properties?

2. What are AVL trees? What are their properties?

3. What are perfectly balanced trees? What are their properties?

4. What is the purpose of Huffman trees?

5. Write an arithmetic expression with at least 5 operations. Build a binary tree for it. Write the infix, prefix, and postfix notations for this expression.

6. Write 5 characters and choose their frequencies of occurrence in a text. Build a Huffman tree for them. Generate codes for the given characters.

7. Write your surname. Encode it using a minimal prefix tree.

8. What is "balancing" of trees? During which operations can a tree become unbalanced?

9. How is balancing of a tree performed?

10. When and at what moment is tree balancing performed?

PRACTICAL ASSIGNMENT 15. DATA FILE TYPES

Goal: Obtain and reinforce knowledge about data types for external data storage.

Preparatory work:

- working with files in programming languages;
- hashing, collision handling;
- trees.

General Information

Logical and physical representation of files. A computer file is a resource for storing data or programs on a computer's permanent storage device. Files have the following characteristics:

- a fixed name that uniquely identifies the file;
- a specific access method (corresponding read and write operations);
- a certain logical structure (organization of stored data).

Characteristics that distinguish one structural element from another can be certain code sequences or simply known values of offsets of these structures relative to the beginning of the file. Data structure support can be entirely delegated to the application or, to some extent, this work can be performed by the file system.

Unstructured file. In the case where all actions related to structuring and interpreting the contents of the file are performed by the application, the file is an unstructured sequence of data. The application sends a request to the file system for input-output, using common facilities for all applications, such as specifying the offset from the beginning of the file and the number of bytes to read or write. The application receives a stream of bytes, which is interpreted according to the logic embedded in the program.

Structured file. In this case, the support for the file's structure is entrusted to the file system. The file system views such a file as an ordered sequence of logical records. The application accesses the file system with requests for input-

output at the level of logical records. In turn, the file system must have information about the file's structure. This approach has evolved into database management systems that support not only complex data structures but also relationships between them.

In programming languages, files are divided into binary and text files based on their logical structure. This division is reflected in the developed applications' data interpretation concept.

Text file. Files intended for storing textual information. Such files consist of a collection of variable-length character strings. The end of each line in such a file is marked by a special "end-of-line" marker. Text files are also characterized by the fact that the information in them has the same appearance as on the screen, meaning it can be read. Accordingly, text files can be read and modified manually using any text editor.

Binary files. These files are used to store data of a specific type. They store information in an internal representation. Such files contain only a sequence of bytes and, therefore, cannot be created or modified manually. However, these inconveniences are compensated by the speed of working with data. In general, binary files can be classified into typed and untyped files. Typed files have read and write units that are associated with a specific data type. Binary files allow direct access to data by directly specifying the address of an element.

A typical task

Create a hashed binary file of a large size organized into sections. Ensure the ability to add and delete records and perform searches based on the key field.

Program code

```
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
#include<string.h>
#include<iostream>
#include <time.h>

#define N 10
#define M 90
#define first
#define G 1000
```

```
struct Tdata
{ int idn;
  int age;
  char name[20];
  int g[150];
};

struct Trazdel
{  Tdata data;
   int free;
   };

FILE *in,*head;
int hsize=40;
Tdata nulldata={0,0,""};

unsigned int hesh(int idn)
{
   return idn % N;
}

int addkey(Trazdel razdel,int adr);
void heshcreate();

void initfile()
{
  int i;
  Trazdel razdel;
  in=fopen("in.bin","wb");
  head=fopen("head.bin","wb");
  for(i=0;i<M;i++)
  { razdel.data.idn=rand()%100;
    razdel.data.age=10 + rand()%10;
    sprintf(razdel.data.name,"%3iname",i);
    razdel.free=1;
    fwrite(&razdel,sizeof(Trazdel),1,in);
         int adr=ftell(in);
         addkey(razdel,adr);
  }
  heshcreate();
  fclose(in);
  fclose(head);
}

void initshow(int f)
 {  int i;
    Trazdel razdel;
    in=fopen("in.bin","rb");
    while(!feof(in))
    {   fread(&razdel,sizeof(Trazdel),1,in);
      //if(f&razdel.free)
        printf("%4d %3d%s ",razdel.data.idn,razdel.data.age,razdel.data.name);
    }
    fclose(in);
 }

void heshcreate()
 {  head=fopen("head.bin","wb");
```

```
    int i,l;
    for(i=0;i<N;i++)
   {   l=i*hsize+N*2;
       fwrite(&l,2,1,head);
   }
   l=-1;
   for(i=0;i<200;i++)
      fwrite(&l,2,1,head);
   fclose(head);
 }

int addkey(Trazdel razdel,int adr)
 {  int key=hesh(razdel.data.idn);
    int i=0;
    int t=0,j;
    fseek(head,key*2,0);
    fread(&key,2,1,head);
    fseek(head,key,0);
    while((i!=-1)&(t<20))
         {    fread(&i,2,1,head);
              t++;
    }
    if(t==20) {printf("Add fail"); return 0;}
    fseek(head,-2,1);
    j=ftell(head);
    fwrite(&adr,2,1,head);
 }

void heshzap()
 {  Trazdel razdel;
    int i=0;
head=fopen("head.bin","r+b");
in=fopen("in.bin","rb");
while(i<M)
{  fread(&razdel,sizeof(Trazdel),1,in);
   addkey(razdel,i*sizeof(Trazdel));
   i++;
}
fclose(head);
fclose(in);
 }

void show()
{  Trazdel razdel;
         head=fopen("head.bin","rb");
         in=fopen("in.bin","rb");
         int key,i=0,j=0;
         int m=0;
         int x=0,lst=0,y=1;
         for(i=0;i<M; i++)//i<10;i++)
         {   j=0;
                  fseek(head,i*2,0);
                  fread(&key,2,1,head);
                  fseek(head,key,0);
                  while((j<20))
                  {   fread(&key,2,1,head);
                      if(key!=-1)
                      {    fseek(in,key,0);
                           fread(&razdel,sizeof(Trazdel),1,in);
```

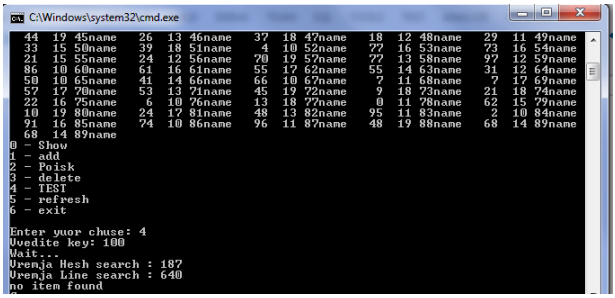
```
                            printf("|%4d %3d
%s\n",razdel.data.idn,razdel.data.age,razdel.data.name);
                            if(++lst==25)
                                     {    lst=0; x++; y=1;}
                                m++;
                            }
                    j++;
      }
   }
}

Tdata hsearch(int key)
 {
  int hsh;
  int j=0;
  hsh=hesh(key);
  Trazdel razdel;
  fseek(head,hsh*2,0);
  fread(&hsh,2,1,head);
  fseek(head,hsh,0);
  while(j<20)
  {  fread(&hsh,2,1,head);
     if(hsh!=-1)
     {   fseek(in,hsh,0);
         fread(&razdel,sizeof(Trazdel),1,in);
         if(razdel.data.idn==key)  return razdel.data;
         j++;
     }
  }
  return nulldata;
}

void add()
 { Trazdel razdel;
   head=fopen("head.bin","r+b");
   in=fopen("in.bin","r+b");
   printf("Vvedite dannie:\nidentifire:");
   std::cin>>razdel.data.idn;
   printf("age:");
   std::cin>>razdel.data.age;
   printf("name:");
   std::cin>>razdel.data.name;
   razdel.free=1;
   if (in != NULL)
   {  fseek(in,0,2);
      int adr=ftell(in);
      addkey(razdel,adr);
      fwrite(&razdel,sizeof(Trazdel),1,in);
   }
   fclose(in);
   fclose(head);
 }

void search()
{  int key;
   printf("Vvedite key:\n");
   std::cin>>key;
   Tdata dat=hsearch(key);
   if(dat.age!=0)
```

```
       printf("Najdeno.\n %4d %3d %s",dat.idn,dat.age,dat.name);
   else printf("Ne najdeno");
   system("pause");
}

void dell()
 {    head=fopen("head.bin","r+b");
         in=fopen("in.bin","r+b");
         int key;
         printf("Vvedite key:\n");
         std::cin>>key;
         int hsh,hsh2;
         int j=0;
         hsh=hesh(key);
         Trazdel razdel;
         fseek(head,hsh*2,0);
         fread(&hsh,2,1,head);
         fseek(head,hsh,0);
         fread(&razdel,sizeof(Trazdel),1,in);
         hsh2=hsh;
         while((j<20)&&(razdel.data.idn!=key))
         {   fread(&hsh,2,1,head);
                  if(hsh!=-1)
                  {     fseek(in,hsh,0);
                        fread(&razdel,sizeof(Trazdel),1,in);
                        j++;
                  }
   }
   if(j==20) printf("No element!");
   else
   {   fseek(in,hsh,0);
       fseek(head,-2,1);
       razdel.free=0;
       fwrite(&razdel,sizeof(Trazdel),1,in);
       hsh2=-1;
       fwrite(&hsh2,2,1,head);
       printf("Del Ok");
    }
  fclose(in);
  fclose(head);
}

void refresh()
 {  initfile();
    heshcreate();
    heshzap();
 }

Tdata lsearch(int key)
  {  Trazdel razdel;
     int j=0;
     fread(&razdel,sizeof(Trazdel),1,in);
     while(!((razdel.free==1)&&(razdel.data.idn==key))&&(j<150))
     {   j++;
         fread(&razdel,sizeof(Trazdel),1,in);
     }
     if(j<150)  return  razdel.data;
     else return nulldata;
  }
```

```
void TEST()
 {  clock_t t1,t2;
  head=fopen("head.bin","rb");
  in=fopen("in.bin","rb");
  long l;
  int key;
  Tdata dat;
  printf("Vvedite key: ");
  std::cin>>key;
  printf("Wait...\n");
  t1=clock();
  for(l=0;l<G;l++)
     dat=hsearch(key);
  t2=clock();
  printf("Vremja Hesh search : %d\n",t2-t1);
  fclose(in);
  fclose(head);
  head=fopen("head.bin","rb");
  in=fopen("in.bin","rb");
  t1=clock();
  for(l=0;l<G;l++)
     dat=lsearch(key);
  t2=clock();
  dat=lsearch(key);
  printf("Vremja Line search : %d\n",t2-t1);
  if(dat.age==0) printf("no item found\n");
  else printf("%3d %3d",dat.idn,dat.age);
  fclose(in);
  fclose(head);
  system("pause");
}

int menu()
{  int k;
   printf("\n0 - Show\n");
   printf("1 - add\n");
   printf("2 - Poisk\n");
   printf("3 - delete\n");
   printf("4 - TEST\n");
   printf("5 - refresh\n");
   printf("6 - exit\n\n");
   printf("Enter yuor chuse: ");
   scanf("%i", &k);
   return k;
}

void main()
{   initfile();
    heshcreate();
    initshow(2);
    int k = menu();
    while(k<7)
   {   switch (k)
       {    case 0: show(); system("pause");; break;
            case 1: add(); show(); system("pause");;break;
            case 2: search(); break;
            case 3: dell(); show(); system("pause");;break;
            case 4: TEST(); break;
```

```
            case 5: refresh(); break;
            case 6:  exit(1); break;
        }
           k = menu();
     }
    system("pause");
}
```

The result of the program:



Individual tasks

Develop a program to work with data stored on disk.

Ensure the following functionalities:

− create a sequential unordered file of initial (input) data, the type of which is determined based on Table 15.1 using the student's number in the journal;

− add and delete records from the file;

− print the contents of the file;

− search for a record using a key.

When performing operations, assume that the file is so large that it cannot be entirely copied into the computer's memory.

Table 15.1 – Individual tasks

N	The type of file being created	Input data content
1	2	3
1	Variable-length sequential file.	Data on workers of a large enterprise
2	Fixed-length sequential file.	Data on countries, cities and towns
3	Hash file (choose the hash function yourself)	Data on the rivers of Eurasia
4	B-tree	Data on phone owners
5	Indexed sequential file	Data about school students
6	Hash file (choose the hash function yourself)	Data from hospital cards
7	Indexed sequential file	Data on the flora of Ukraine
8	Fixed-length sequential file.	Data about students of the institute
9	B-tree	Data on phone owners
10	B+ tree	Data on the fauna of Ukraine
11	Hash file (choose the hash function yourself)	Data on the availability of seats in trains in all directions
12	B-tree	Library fund data
13	Variable-length sequential file.	Statistical data on the harvest of agricultural crops for N years
14	B+ tree	Synoptic indicators for all years of observation

Note: N is the student's number in the journal.

Checkpoint questions

1. What does the term "Physical file organization" mean?
2. What is "disk space fragmentation"?
3. What are the most commonly used data placement schemes in files at the physical level?
4. What is a "block" of a file?
5. How are data recorded in contiguous files?
6. How are data recorded in files represented by a linked list of blocks?
7. How are data recorded in files represented by a linked list of block indexes?
8. What types of files are distinguished at the logical level?
9. What are byte-oriented files?
10. What are structured files?
11. What form can the logical structure of direct access files take?
12. What is the structure of hashed files?
13. How is a file created based on the B-tree principle?
14. What is the difference between files created based on the B+-tree principle and files created based on the B-tree principle?

PRACTICAL ASSIGNMENT 16. ALGORITHMS OF EXTERNAL SORTING

GOAL: ACQUIRE AND CONSOLIDATE SKILLS IN SORTING THE CONTENT OF VERY LARGE FILES.

Preparatory work:

- search algorithms;
- internal sorting algorithms;
- physical and logical file structure;
- binary files;
- sequential and direct access files.

Types of external sorting algorithms

Main concepts. *External sorting* is the sorting of data that is located on external devices and does not fit into the main memory. Among the most well-known algorithms of external sorting are:

- merge sort (simple merge and natural merge);
- improved sorting (multi-phase and cascade sorting).

The fundamental concept when using external sorting is the concept of a run. *A run* (ordered segment) is a sequence of elements that is ordered by a key. The number of elements in a run is called the run's length. A run consisting of a single element is always ordered. The last run may have a length smaller than the rest of the runs in the file. The maximum number of runs in a file is N (all elements are unsorted). The minimum number of runs is one (all elements are sorted).

Most of the methods of external sorting are based on the procedures of merging and distribution. *Merging* is the process of combining two (or more) ordered runs into a single ordered sequence using a cyclic selection of elements available at that moment. *Distribution* is the process of distributing ordered runs into two or several auxiliary files. A phase is an action performed in a single pass over the entire sequence of elements. *Two-phase sorting* is a sorting

method in which two phases, distribution and merging, are implemented separately. *One-phase sorting* is a sorting method in which the distribution and merging phases are combined into one. *Two-way* merging refers to sorting where the data is distributed into two auxiliary files. *Multi-way merging* refers to sorting where the data is distributed into N (N > 2) auxiliary files.

Direct merge algorithm. Let there be a sequential file *A*, consisting of records a_1, a_2, ..., a_n (for simplicity, let's assume that *n* is a power of 2). Two auxiliary files B and C are used for sorting, each with a size of *n/2*.

The sorting process consists of a sequence of steps, each of which involves distributing the contents of file *A* into files *B* and *C*, followed by merging files *B* and *C* back into file *A*. In the first step, file *A* is read sequentially, and records a_1, a_3, ..., $a_{(n-1)}$ are written to file *B*, while records a_2, a_4, ..., *an* are written to file *C* (initial distribution).

The initial merging is performed over pairs (a_1, a_2), (a_3, a_4), ..., $(a_{(n-1)}, an)$, and the result is written to file *A*. In the second step, file *A* is sequentially read again, and in file *B*, sequential pairs with odd numbers are written, while in file *C*, pairs with even numbers are written. During merging, ordered sets of four records are formed and written back to file *A*. And so on. Before performing the last step, file *A* will contain two ordered subsequences, each with a size of n/2. In the distribution step, the first subsequence will go to file *B*, and the second subsequence will go to file *C*. After merging, file *A* will contain a completely ordered sequence of records.

Natural merge sort. When using the natural merge sort method, it does not take into account that the original file may be partially sorted, that is, it may contain ordered subsequences of records - sequences. The natural merge sort method is based on recognizing sequences during distribution and using them during the subsequent merge.

As in the case of direct merge sort, sorting is performed in several steps, in each of which the file A is first distributed into files B and C, and then B and C are merged into file A. During distribution, the first sequence of records is recognized and written to file B, the second - to file C, and so on. During merging, the first sequence of file *B* is merged with the first sequence of file *C*, the second sequence of *B* - with the second sequence of *C*, and so on. If one file is completely read before the other (due to a different number of sequences), the

remaining unprocessed part of the file is copied in its entirety to the end of file *A*. The process is completed when only one sequence remains in file *A*.

Multi-phase sorting. The idea of multi-phase sorting is that out of the available *m* auxiliary files, *(m-1)* files are used for inputting sequences to be merged, and one is used for outputting the generated runs. As soon as one of the input files becomes empty, it is then used for outputting runs obtained from the merging of runs from the new set of *(m-1)* files. Thus, in the first step, the runs of the initial file are distributed among *m-1* auxiliary files, and then in the subsequent steps, multi-way merging of runs from *(m-1)* files is performed until one of them contains only one run.

It has been proven that the three-phase external sorting method produces the desired result and works most efficiently (merging the maximum number of runs at each stage) if the initial distribution of runs among auxiliary files is described by neighboring Fibonacci numbers.

In general, when using m auxiliary files, the condition for successful completion and efficient operation of the multi-phase external sorting method is that the initial distribution of runs among *m-1* files is described by the sums of neighboring *(m-1), (m-2), ..., 1* Fibonacci numbers of order *m-2*. The sequence of Fibonacci numbers of order p starts with *(p-1)* zeros, the p-th element is 1, and each subsequent element is the sum of the previous p elements. For example, here is a sequence of Fibonacci numbers of order 5: 0, 0, 0, 0, 1, 1, 2, 4, 8, 16, 31, 61, ...

It is clear that if the distribution is based on the Fibonacci number *fi*, then the minimum number of runs in the auxiliary files is equal to *fi*, and the maximum is *f(i+1)*. Therefore, after performing the merge, the maximum number of runs is *fi*, and the minimum *f(i-1)* is obtained. At each stage, the maximum possible number of merges is performed, and the process reduces to having only one run.

Since the number of series in the source file may not provide the possibility of such distribution of series, the method of adding empty series is used, which are further distributed as evenly as possible between intermediate files and oriented during subsequent mergers. It is clear that the fewer such empty series, the closer the number of initial series is to the requirements of Fibonacci and the more efficiently the algorithm works.

Improving the efficiency of external sorting. It is clear that the longer the sequences the file contains before the application of external sorting, the fewer merges will be required, and the sorting finishes more quickly. Therefore, before applying any of the external sorting methods based on the use of sequences, the initial file is read into the main memory in parts, and one of the most efficient internal sorting algorithms (for example, Quicksort or Heapsort) is applied to each part. The sorted parts, forming sequences, are then written to a new file (the old one cannot be used).

A typical task

Implement an external sorting algorithm. Check performance.

Explanation to the text of the program. The initial file input.txt contains 1000 random numbers. 10 temporary files are created, in each of which approximately 10 parts of the contents of the initial file are overwritten. Their content is sorted by an internal sorting algorithm.

Program code

```
#include <queue>
#include <fstream>
#include <algorithm>
#include <climits>
#include <ctime>
#include <stdlib.h>
#define chunk_count 10

struct MinHeapNode
{ int element;
  int index;
};
struct Comparator
{  bool operator() (const MinHeapNode lhs, const MinHeapNode rhs) const
  { return lhs.element > rhs.element;
  }
};

void create_chunks(std::string filename, std::string outfilename,int chunk_size,
int achunk_count);
void merge_chunks(std::string filename, int chunk_size, int achunk_count);
void generate_input_file(std::string filename, int length);

int main()
{ int chunk_size = 1000;
  std::string input_filename = "input.txt";
  std::string output_filename = "output.txt";
  generate_input_file(input_filename, chunk_count * chunk_size);
  create_chunks(input_filename, output_filename, chunk_size, chunk_count);
  merge_chunks((char*)output_filename.c_str(), chunk_size, chunk_count);
```

```
  return 0;
}

void merge_chunks(std::string filename, int chunk_size, int achunk_count)
{ char str[10];
  std::ifstream input_files[chunk_count];
  for (int i = 0; i < achunk_count; i++)
  {  _itoa_s(i, str, 10);
     input_files[i] = std::ifstream(str);
  }
  std::ofstream output_file(filename);
  MinHeapNode harr[chunk_count];
  std::priority_queue<MinHeapNode, std::vector<MinHeapNode>, Comparator> pq;
  int i;
  for (i = 0; i < achunk_count; i++)
  { if (!(input_files[i] >> harr[i].element))
    break;
    harr[i].index = i;
    pq.push(harr[i]);
  }
  int count = 0;
  while (count != i)
  { MinHeapNode root = pq.top();
    q.pop();
    output_file << root.element << ' ';
    if (!(input_files[root.index] >> root.element))
    {  root.element = INT_MAX;
       count++;
    }
    pq.push(root);
  }
  for (int i = 0; i < chunk_count; i++)
    input_files[i].close();
  output_file.close();
}
void create_chunks(std::string filename, std::string outfilename, int chunk_size,
int achunk_count)
{ std::ifstream input_file(filename);
  char str[10];
  std::ofstream output_files[chunk_count];
  for (int i = 0; i < achunk_count; ++i)
  { _itoa_s(i, str, 10);
    output_files[i] = std::ofstream(str);
  }
  int* arr = new int[chunk_size];
  int output_file_index = 0;
  bool is_finised = false;
  while (!is_finised)
  { int current_chunk_size = 0;
    while (current_chunk_size < chunk_size)
    { if (!(input_file >> arr[current_chunk_size]))
      { is_finised = true;
        break;
      } else
       ++current_chunk_size;
    }
    std::sort(arr, arr + current_chunk_size);
    for (int i = 0; i < current_chunk_size; i++)
    { output_files[output_file_index] << arr[i] << ' ';
```

```
    }
    ++output_file_index;
  }
  for (int i = 0; i < chunk_count; i++)
    output_files[i].close();
  input_file.close();
}
void generate_input_file(std::string filename, int length)
{ std::ofstream input_file(filename);
  std::srand(time(NULL));
  for (int i = 0; i < length; i++)
    input_file << rand() << ' ';
  input_file.close();
}
```

The result of the program:

The composition of the created files is shown in fig. 16.1, the content of the initial file is shown in fig. 16.2, the contents of the temporary files 0.txt and 9.txt - in fig. 16.3 and 16.4, accordingly, the content of the file with ordered data is shown in fig. 16.5.

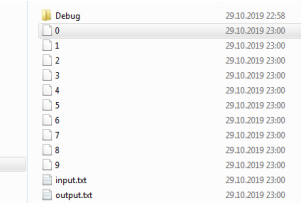


Figure 16.1 – Project text files

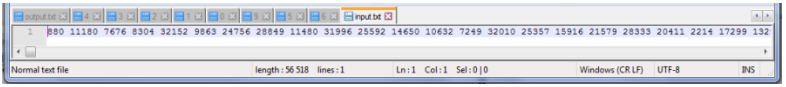


Figure 16.2 – Contents of the initial file input.txt (56 KB)

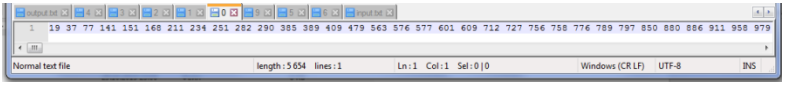


Figure 16.3 – Contents of temporary file 0.txt (6 KB)

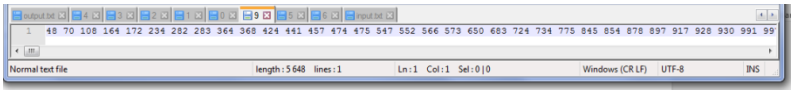


Figure 16.4 – Contents of temporary file 9.txt (6 KB)

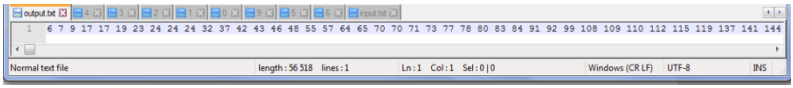


Figure 16.5 – Content of the resulting output.txt file (56 KB)

Individual tasks

Develop and debug a program to implement the external sorting algorithm (Table 16.1).

Table 16.1 – Individual tasks

N	The name of the external sorting algorithm
1	Direct merge, 3 files.
2	Natural merge, 4 files. For creating series - Quicksort algorithm.
3	Multi-phase merge, 6 files. For series - Quicksort algorithm.
4	Balanced multiway merge, 6 files. For series - Quicksort algorithm.
5	Direct merge, 3 files.
6	Natural merge, 3 files. For creating series - Heapsort algorithm.
7	Multi-phase merge, 8 files. For series - Heapsort algorithm.
8	Balanced multiway merge, 8 files. For series - Heapsort algorithm.
9	Multi-phase merge, 10 files. For series - Quicksort algorithm.
10	Balanced multiway merge, 10 files. For series - Quicksort algorithm.
11	Natural merge, 3 files. For creating series - Quicksort algorithm.
12	Multi-phase merge, 10 files. For series - Heapsort algorithm.
13	Balanced multiway merge, 10 files. For series - Heapsort algorithm.

Checkpoint questions

1. What is the fundamental difference between external sorting algorithms and internal sorting algorithms?
2. Provide a description of the direct merge algorithm.
3. What are the phases involved in direct merge sorting?
4. How many files are used in the direct merge algorithm?
5. Describe the algorithm for natural merge sorting.
6. Can partial ordering of the initial file be taken into account?
7. How many files can be used in the natural merge algorithm?
8. How many files can be used in multi-phase merge sorting?
9. Provide a description of the multi-phase merge algorithm.
10. What does it mean when an algorithm fails to converge in sorting? How can this situation be resolved?

BIBLIOGRAPHY

1. Introduction to algorithms, T. Kormen, Ch. Leizerson, R. Ryvest, K. Shtain. 2nd edition.: Translated from English – M: "Williams" Publishing House, 2007 – 1290 p.

2. Stroustrup, Bjarne. The C++ Programming Language. Special edition, /B. Stroustrup; translated from English; edited by N. N. Martynov. Vol. 1: Fundamental Algorithms. – M: Binom, 2011. – 1035 p.

3. Knuth, Donald Ervin. The Art of Computer Programming /D. A. Knuth; Translated from English; – M: "Williams" Publishing House, 2007. – 3rd edition. – 712 p.

4. Knuth, Donald Ervin. The Art of Computer Programming, : vol. 2: Seminumerical Algorithms /D. E. Knuth, Translated from English: : "Williams" Publishing House, – 3rd edition, 2007. – M. 828 p.

5. Knuth, Donald Ervin. The Art of Computer Programming: vol. 3: Sorting and Searching. /D. E. Knuth; Translated from English. – M: "Williams" Publishing House, 2007. 3rd edition. – 822 p.

6. Aho, A. V., Hopcroft, J., Ullman, J. D. Data Structures and Algorithms. /A. V. Aho, J. E. Hopcroft, J. D. Ullman; Translated from English. – M: "Williams" Publishing House, 2003. – 384 p.

7. Sedgewick, R. Fundamental Algorithms in C++, : in 5 parts. Parts 1–4: Analysis / Data Structures / Sorting / Searching / R. Sedgewick; Translated from English. – Kyiv: "DiaSoft" Publishing House, 2001. – 688 p.

8. Sedgewick, R. Fundamental Algorithms in C++: in 5 parts. Part 5. Graph Algorithms / R. Sedgewick; – St. Petersburg: "DiaSoftUP", 2002. – 496 p.

.

FOR NOTES

Educational-methodical publication

БУЛЬБА Сергій Сергійович
БРЕЧКО Вероніка Олександрівна
КУЧУК Ніна Георгіївна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчально-методичний посібник
для студентів спец. 123
«Комп'ютерна інженерія».

Відповідальний за випуск проф. Олександр ЗАКОВОРОТНИЙ
Роботу до видання рекомендував проф. Микола ЗАПОЛОВСЬКИЙ

План 2024 р., поз. 26

Видавець:
Видавничий центр НТУ «ХПІ»
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Фрунзе, 21

Самостійне електронне видання